P25M User's Manual

P25M User's Manual

The P25M User's Manual was prepared by the technical staff of Innovative Integration on February 5, 2009.

For further assistance contact:

Innovative Integration
2390-A Ward Ave
Simi Valley, California 93065

PH: (805) 578-4260
FAX: (805) 578-4225

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

This document is copyright 2009 by Innovative Integration. All rights are reserved.

VSS \ Distributions \ P25M \ Documentation \ Manual \ P25MMaster.odm

#XXXXXX

Rev 1.0

Table of Contents

Introduction....10
Real Time Solutions!....10
Vocabulary....10
What is the P25M? 10
What is Malibu? 11
What is C++ Builder?....11
What is Microsoft MSVC?....11
What kinds of applications are possible with Innovative Integration hardware?....11
Why do I need to use Malibu with my Baseboard?....11
Finding detailed information on Malibu....12
Online Help....12
Innovative Integration Technical Support....12
Innovative Integration Web Site....12
Typographic Conventions....12
Installation....14
Host Hardware Requirements....14
Software Installation....14
Tools Registration....17
Bus Master Memory Reservation Applet....17
Hardware Installation....18
JTAG Emulator Hardware Installation (for DSP boards Only)....18
PCI Pod-Based Emulator Installation....18
Baseboard Installation....19
A Few Considerations BEFORE Power-up....19
It cannot be overemphasized: 19
After Power-up....20
Code Composer Studio Setup....20
Borland Builder Setup and Use....22
Automatic saving of project files and forms during debugging....23
Static-binding of built executables. 23
Appropriate library and include paths. 24
About the Baseboard....25
DSP Baseboard Hardware Features....25
Digital Signal Processor....26
DSP External Memory....26
DSP Initialization....26
DSP JTAG Debugger Support....27
FPGA Core....27
FPGA Memory....28
FPGA Clocks....29
Custom Logic Applications....29
Loading the Application Logic....29
FPGA Initialization....29
The Pismo Class Library....30
Simple To Use....30

Not Just for C++ Experts 31
Analog I/O Streams....32
Stream Objects and Device Drivers....32
Hardware Isolation and Independence....32
Stream I/O Types....32
Stream Buffer Model....33
Stream Internals....34
Multitasking Friendly....34
Using Analog Streams in an Application 34
Selecting the Stream Object....35
Selecting and Configuring Hardware....36
Selecting and Configuring Clocks....36
Selecting and Configuring Triggers....37
Selecting Pretriggering Modes....37
Interrupt Handling....39
Interrupts in a C++ Environment....39
The Pismo Solution....40
Class Irq....41
Interrupt Lock Classes....41
Interrupt Binder Templates 41
Class InterruptHandler....41
Class ClassMemberHandler Template....41
Class FunctionHandler Template....42
EDMA and QDMA Handling....43
Class DmaSettings....43
Class Qdma....43
Class Edma....45
Linked and Chained blocks....47
Class EdmaMaster....47
P25 Example Programs....48
The Next Step: Developing Custom Code....49
Peripherals....50
Analog Input....50
Analog Input Characteristics....50
Programmable Gain....51
Analog Input Bandwidth....51
A/D Device 52
Calibration and Error Correction....53
A/D FIFO....54
A/D Sample Rate Restrictions....54
Analog Output Characteristics....55
Analog Output Bandwidth....56
DAC Device 57
Calibration and Error Correction....58
DAC FIFO....58
DAC Sample Rate Restrictions....59
Triggering 59
Framed Trigger Mode....60
Decimation....60
Digital I/O....61

Software Support....61
Hardware Implementation....61
Digital I/O Timing....63
Digital IO Electrical Characteristics....63
Sample Rate Generation and Clocking Controls....64
External Clock Input....65
Generating a Sample Clock with the PLL....65
Fixed Oscillator Option for Sample Clock....67
PLL Status....68
PLL Interface....68
Notes About Programming the PLL....69
Serial EEPROM Interface....70

Servo Applications....71

Data Acquisition Applications....71
Servo Control Applications....71
Servoing and DSP/BIOS....71
The Servo Base Class....72
Using the Servo Class....72
Servo Interrupt Modes....72
Execute()....73
Timebases....73
DAC Delay....73
Channel Selection....73
Servo Timing....74
A Servo Tutorial....75
Conclusion....78

Communication to the Host....79

Overview....79
CPU Busmastering Interface....79
CPU Busmastering Implementation....80
Packet Based Transfers....80
Blocking Interface....80
Maximum Transfer Size....80
Host (Armada Library) Support for CPU Busmastering....80
Target (Pismo Library) Support for CPU Busmastering....81
C++ Terminal I/O....81
Target Software....82
Tutorial....82

Developing Host Code....84

The Malibu Libraries....84
Malibu Library Features....85
Event Handlers....85
The P25M in the Host Environment....86
Connection to the Hardware....86
COFF loading to the 6713 Processor....86
Target / Host Packet Communication....86
Scope Host Program....88
DspBoardFtns.cpp Functions....89

Host Example Programs for the P25M Baseboard....91
Overview....91
Command and Data Packets....91
ASnap....91
DataXfer....92
Building a Target DSP Project....93
Writing a Program....98
Host Tools for Target Application Development....98
Components of Target Code (.cpp, .tcf, .cmd, .pjt)....98
Edit-Compile-Test Cycle using Code Composer Studio....99
Automatic projectfile creation....99
Rebuilding a Project....99
IIMain replaces main....99
Running the Target Executable....99
Note:....100
Anatomy of a Target Program....100
Use of Library Code....101
Example Programs....101
The Next Step: Developing Custom Code....102
Developing Host Applications....103
Borland Turbo C++....103
Microsoft Visual Studio 2005....105
Applets....107
Common Applets....107
Registration Utility (NewUser.exe)....107
ReserveMemoryDsp....108
....108
Data Analysis Applets....108
Binary File Viewer Utility (BinView.exe)....108
Target Programming Applets....109
Target Project Copy Utility (CcsProjectCopy.exe)....109
Demangle Utility (Demangle.exe)....109
COFF Section Dump Utility (CoffDump.exe)....109
JTAG Diagnostic Utility (JtagDiag.exe)....110
DEF Conversion Utility (DefConvert.exe)....110
PCI Configuration Utility (Config.exe)....111
RtdxTerminal - Terminal Emulator....111
Important Note:....112
Terminal Emulator Menu Commands....112
The File Menu....112
The DSP Menu....112
The Form Menu....113
The Help Menu....113
Options Tab:....113
Display Group....114
Sounds Group....115
Coff Load Group....115
Debugger Group....115

Terminal Emulator Command Line Switches....115
Baseboard-specific Applets....116
EEProm (EEProm.exe)....116
Finder (Finder.exe)....116
Target Number....116
Blink....116
On/OFF....117
Logic Loader (LogicLoader.exe)....117
Connector Pinout and Physical Information....118
P25M Connector Pinouts....118
Analog Input/Output Connectors....118
J6 – External Clock Input/Output....118
J5 – External Sync/Trigger....120
JP3 – Digital I/O Connector....122
JP4 – UD Digital I/O Connector....123
JP8 – DSP JTAG....124
JP5 – FPGA JTAG Connector....124
Board Layout....126

List of Tables

Table 1. ‘C6713 DSP EMIF Control Register Initialization Values....26
Table 2. Device Driver and Stream Classes....32
Table 3. Stream Object Clock Methods....36
Table 4. Stream object Pretrigger Methods....37
Table 5. Stream object Start Trigger Methods....37
Table 6. Stream object Stop Trigger Methods....38
Table 7. Stream object Retrigger Methods....38
Table 8. Interrupt Lock Classes....41
Table 9. P25 Example Programs....48
Table 10. Standard Analog Input Ranges....51
Table 11. Programmable Gain Settings....51
Table 12. A/D output codes....52
Table 13. A/D Configuration Settings....53
Table 14. Analog Input Error Correction Coefficients....53
Table 15. A/D Data Ordering in the FIFO....54
Table 16. DAC Output Characteristics....56
Table 17. DAC output codes....57
Table 18. Analog Output Error Correction Coefficients....58
Table 19. DAC Data Ordering in the FIFO....58
Table 20. DigitalIo Class Operations....61
Table 21. Digital I/O Port Timing Parameters....63
Table 22. Digital IO Electrical Characteristics....64
Table 23. P25M External Clock Input Requirements....65
Table 24. PLL Interface Word Format....68
Table 25. PLL Read Sequence....68
Table 26. PLL Read Word....69
Table 1. PLL Output Assignments....69
Table 27. The Servo Class Virtual Function....72
Table 28. Servo Time Measurements....74
Table 29. Some Classes In Malibu....85
Table 30. Pismo Example Programs....102
Table 31. DSP Serial Port Connectors....120

List of Figures

Figure 1. A/D Signal Flow....50
Figure 2. A/D Input Frequency Response....52
Figure 3. Output Signal Flow....55
Figure 4. D/A Signal Flow....55
Figure 5. DAC Output Filter Frequency Response....57
Figure 6. DAC Output Filter Phase Response....57
Figure 7. Analog Triggering Timing....60
Figure 8. DIO Control Register (0x80000000)....62
Figure 9. UD Digital I/O Configuration Control Register (0x802A0000)....62
Figure 10. UD Digital I/O Configuration Control Register Definition (0x802A0000)....62
Figure 11. Digital IO Port Addresses....62
Figure 12. Digital I/O Port Timing....63
Figure 13. P25M Sample Clock Generation and Distribution Block Diagram....64
Figure 1. PLL Reference Prescaling....66
Figure 2. Examples of PLL Output Frequencies....67
Figure 14. RtdxTerminal Options....114
Figure 15. JP6 and JP7 DSP McBSP Serial Port Connector Pinouts....121
Figure 16. JP6 and JP7 DSP McBSP Serial Port Connector Pin Orientation....121
Figure 17. Digital I/O Connector Pinout....122
Figure 18. JP3 Digital I/O Connector Pin Orientation....122
Figure 19. JP4 UD Digital I/O Connector Pinout....123
Figure 20. JP4 UD Digital I/O Connector Pin Orientation....123
Figure 21. JP8 DSPJTAG Connector Pinouts....124
Figure 22. JP8 DSP JTAG Connector Pin Orientation....124
Figure 23. JP5 FPGA JTAG Connector Pinouts....125
Figure 24. JP5 FPGA JTAG Connector Pin Orientation....125

Introduction

Real Time Solutions!

Thank you for choosing Innovative Integration, we appreciate your business! Since 1988, Innovative Integration has grown to become one of the world's leading suppliers of DSP and data acquisition solutions. Innovative offers a product portfolio unrivaled in its depth and its range of performance and I/O capabilities .

Whether you are seeking a simple DSP development platform or a complex, multiprocessor, multichannel data acquisition system, Innovative Integration has the solution. To enhance your productivity, our hardware products are supported by comprehensive software libraries and device drivers providing optimal performance and maximum portability.

Innovative Integration's products employ the latest digital signal processor technology thereby providing you the competitive edge so critical in today's global markets. Using our powerful data acquisition and DSP products allows you to incorporate leading-edge technology into your system without the risk normally associated with advanced product development. Your efforts are channeled into the area you know best ... your application.

Vocabulary

What is the P25M?
The P25M is Innovative Integration's PCI plug-in DSP card that integrates high-performance analog and digital peripherals with a high-performance DSP and FPGA core. The P25M is equipped with a 64-bit, 66 MHz PCI bus interface allowing high speed data transfers with the host.

The P25M includes an on-board Texas Instruments TMS320C6713 DSP with 128 MB cached SDRAM and a 1M gate Xilinx Spartan3 FPGA with 2MB local memory. The DSP and FPGA form the powerful computing core at the heart of the P25M. The P25M has 4 channels of 25 MSPS, 16-bit A/D and four channels of 50 MSPS 16-bit DAC, as well as 64 bits of digital IO that directly connect to the FPGA for high speed signal processing. The P25M PCI bus interface communicates with the host at speeds up to 512 MB/sec using a packet-message-based bus interface. With the on-bard combination of the DSP/FPGA and analog IO, the P25M is capable of performing high speed data collection, servo or other real-time signal processing as an independent signal processing node.

What is Malibu?
Malibu is the Innovative Integration-authored component suite, which combines with the Borland BCB or Microsoft MSVC Integrated Development Environments (IDEs) to support baseboard programming. Malibu supports both high-speed data streaming plus asynchronous mailbox communications between the DSP and the Host PC, plus a wealth of Host functions to visualize and post-process data received from or to be sent to the target DSP.

What is C++ Builder?
C++ Builder is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the Builder IDE through the addition of functional blocks (VCL components) specifically tailored to perform real-time data streaming functions.

What is Microsoft MSVC?
MSVC is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the MSVC IDE through the addition of dynamically created MSVC-compatible C++ classes specifically tailored to perform real-time data streaming functions.

What kinds of applications are possible with Innovative Integration hardware?
Data acquisition, data logging, stimulus-response and signal processing jobs are easily solved with Innovative Integration baseboards using the Malibu software. There are a wide selection of peripheral devices available in the Matador DSP product family, for all types of signals from DC to RF frequency applications, video or audio processing. Additionally, multiple Innovative Integration baseboards can be used for a large channel or mixed requirement systems and data acquisition cards from Innovative can be integrated with Innovative's other DSP or data acquisition baseboards for high-performance signal processing.

Why do I need to use Malibu with my Baseboard?
One of the biggest issues in using the personal computer for data collection, control, and communications applications is the relatively poor real-time performance associated with the system. Despite the high computational power of the PC, it cannot reliably respond to real-time events at rates much faster than a few hundred hertz. The PC is really best at processing data, not collecting it. In fact, most modern operating systems like Windows are simply not focused on real-time performance, but rather on ease of use and convenience. Word processing and spreadsheets are simply not high-performance real-time tasks.

The solution to this problem is to provide specialized hardware assistance responsible solely for real- time tasks. Much the same as a dedicated video subsystem is required for adequate display performance, dedicated hardware for real-time data collection and signal processing is needed. This is precisely the focus of our baseboards – a high performance, state-of-the-art, dedicated digital signal processor coupled with real-time data I/O capable of flowing data via a 64-bit PCI bus interface.

The hardware is really only half the story. The other half is the Malibu software toolset which uses state of the art software techniques to bring our baseboards to life in the Windows environment. These software tools allow you to create applications for your baseboard that encompass the whole job - from high speed data acquisition, to the user interface

Finding detailed information on Malibu
Information on Malibu is available in a variety of forms:

- On-line Help
- Innovative Integration Technical Support
- Innovative Integration Web Site (www.innovative-dsp.com)

Online Help

The online help system for Malibu is fully integrated into the excellent OpenHelp system provided with Builder. Help for Malibu is provided in a single file, Malibu.hlp which is installed beneath the main Builder C++ directory tree during the default installation. It provides detailed information about the components contained in Malibu - their Properties, Methods, Events, and usage examples. An equivalent version of this help file in HTML help format is also provided: Malibu.chm, for use within the MSVC context.

Innovative Integration Technical Support

Innovative includes a variety of technical support facilities as part of the Malibu toolset. Telephone hotline supported is available via

Hotline (805) 578-4260 8:00AM-5:00 PM PST.

Alternately, you may e-mail your technical questions at any time to:

techsprt@innovative-dsp.com .

Innovative Integration Web Site

Additional information on Innovative Integration hardware and the Malibu Toolset is available via the Innovative Integration website at www.innovative-dsp.com

Typographic Conventions

This manual uses the typefaces described below to indicate special text.

Typeface	Meaning
`Source Listing`	Text in this style represents text as it appears onscreen or in code. It also represents anything you must type.
Boldface	Text in this style is used to strongly emphasize certain words.
Emphasis	Text in this style is used to emphasize certain words, such as new terms.
Cpp Variable	Text in this style represents C++ variables
Cpp Symbol	Text in this style represents C++ identifiers, such as class, function, or type names.
KEYCAPS	Text in this style indicates a key on your keyboard. For example, "Press **ESC** to exit a menu".
Menu Command	Text in this style represents menu commands. For example "Click View | Tools | Customize"

Windows Installation

This chapter describes the software and hardware installation procedure for the Windows platform (WindowsXP and Vista).

> **Do *NOT*** install the hardware card into your system at this time. This will follow the software installation.

Host Hardware Requirements

The software development tools require an IBM or 100% compatible Pentium IV - class or higher machine for proper operation. An Intel-brand processor CPU is *strongly recommended*, since AMD and other "clone" processors are not guaranteed to be compatible with the Intel MMX and SIMD instruction-set extensions which the Armada and Malibu Host libraries utilize extensively to improve processing performance within a number of its components. The host system must have at least 128 Mbytes of memory (256MB recommended), 100 Mbytes available hard disk space, and a DVD-ROM drive. Windows2000 or WindowsXP (referred to herein simply as *Windows*) is required to run the developer's package software, and are the target operating systems for which host software development is supported.

Software Installation

The development package installation program will guide you through the installation process.

> **Note:** Before installing the host development libraries (VCL components or MFC classes), you must have Microsoft MSVC Studio (version 9 or later) and/or Codegear RAD Studio C++ (version 11) installed on your system, depending on which of these IDEs you plan to use for Host development. If you are planning on using these environments, it is imperative that they are tested and known-operational before proceeding with the library installation. If these items are not installed prior to running the Innovative Integration install, the installation program will not permit installation of the associated development libraries. However, drivers and DLLs may be installed to facilitate field deployment.

You must have **Administrator Privileges** to install and run the software/hardware onto your system, refer to the Windows documentation for details on how to get these privileges.

Starting the Installation

To begin the installation, start Windows. Shut down all running programs and disable anti-virus software. Insert the installation **DVD**. If Autostart is enabled on your system, the install program will launch. If the DVD does not Autostart, click on Start | Run... Enter the path to the **Setup.bat** program located at the root of your DVD-ROM drive (i.e. **E:\Setup.bat**) and click "OK" to launch the setup program.

> **SETUP.BAT** detects if the OS is 64-bit or 32-bit and runs the appropriate installation for each environment. It is important that this script be run to launch an install.

When installing on a Vista OS, the dialog below may pop up. In each case, select "Install this driver software anyway" to continue.

Windows Security
Windows can't verify the publisher of this driver software
Don't install this driver software
You should check your manufacturer's website for updated driver software for your device.
Install this driver software anyway
Only install driver software obtained from your manufacturer's website or disc. Unsigned software from other sources may harm your computer or steal information.
See details

Figure 1. Vista Verificaton Dialog

The Installer Program

After launching Setup, you will be presented with the following screen.

Figure 2. Innovative Install Program

Using this interface, specify which product to install, and where on your system to install it.

1) Select the appropriate product from the Product Menu.

2) Specify the path where the development package files are to be installed. You may type a path or click "**Change**" to browse for, or create, a directory. If left unchanged, the install will use the default location of "C:\Innovative".

3) Typically, most users will perform a "Full Install" by leaving all items in the "**Components to Install**" box checked. If you do not wish to install a particular item, simply uncheck it. The Installer will alert you and automatically uncheck any item that requires a development environment that is not detected on your system.

4) Click the Install button to begin the installation.

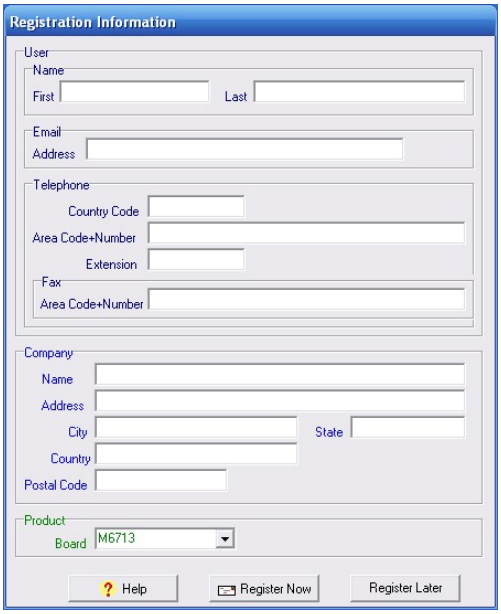
> **Note:** The default "Product Filter" setting for the installer interface is "Current Only" as indicated by the combo box located at the top right of the screen. If the install that you require does not appear in the "Product Selection Box" (1), Change the "Product Filter" to "Current plus Legacy".

Each item of the checklist in the screen shown above, has a sub-install associated with it and will open a sub-install screen if checked. For example, the first sub-install for “Quadia - Applets, Examples, Docs, and Pismo libraries” is shown below.

The installation will display a progress window, similar to the one shown below, for each item checked.

Figure 3. Progress is shown for each section.

Tools Registration



At the end of the installation process you will be prompted to register. If you decide that you would like to register at a later time, click "Register Later".

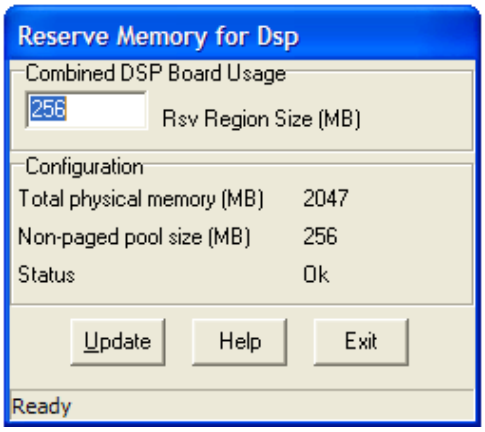
When you are ready to register, click Start | All Programs | Innovative | <Board Name> | Applets. Open the New User folder and launch **NewUser.exe** to start the registration application. The registration form to the left will be displayed.

Before beginning DSP and Host software development, you must register your installation with Innovative Integration. Technical support will not be provided until registration is successfully completed. Additionally, some development applets will not operate until unlocked with a passcode provided during the registration process.

It is recommend that you completely fill out this form and return it to Innovative Integration, via email or fax. Upon receipt, Innovative Integration will provide access codes to enable technical support and unrestricted access to applets.

Figure 4. ToolSet registration form

Bus Master Memory Reservation Applet.



At the conclusion of the installation process, **ReserveMem.exe** will run (except for SBC products). This will allow you to set the memory size needed for the busmastering to occur properly. This applet may be run from the start menu later if you need to change the parameters.

For optimum performance each Matador Family Baseboard requires 2 MB of memory to be reserved for its use. To reserve this memory, the registry must be updated using the ReserveMem applet. Simply select the **Number of Baseboards** you have on your system, click **Update** and the applet will update the registry for you. If at any time you change the number of boards in your system, then you must invoke this applet found in Start | All Programs | Innovative | <target board> | Applets | Reserve Memory.

After updating the system exit the applet by clicking the **exit** button to resume the installation process.

Figure 5. BusMaster configuration

At the end of the install process, the following screen will appear.

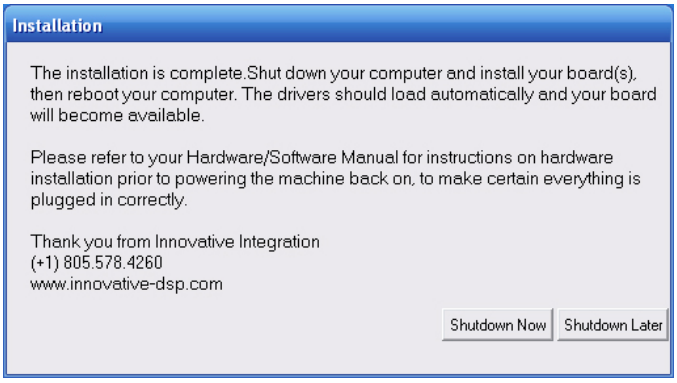

Installation
The installation is complete.Shut down your computer and install your board(s), then reboot your computer. The drivers should load automatically and your board will become available.
Please refer to your Hardware/Software Manual for instructions on hardware installation prior to powering the machine back on, to make certain everything is plugged in correctly.
Thank you from Innovative Integration
(+1) 805.578.4260
www.innovative-dsp.com
Shutdown Now
Shutdown Later

Figure 6. Installation complete

Click the "Shutdown Now" button to shut down your computer. Once the shutdown process is complete unplug the system power cord from the power outlet and proceed to the next section, "Hardware Installation."

Hardware Installation

Now that the software components of the Development Package have been installed the next step is to configure and install your hardware. Detailed instructions on board installation are given in the Hardware Installation chapter, following this chapter.

> **IMPORTANT:** Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

After Power-up

After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the "TroubleShooting" section.

JTAG Hardware Installation

JTAG Emulator Hardware Installation (for DSP boards Only)

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	Features
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including 'C54x and 'C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

5) Perform the board installation in an “ESD” or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

6) Shut down Windows, power-off the host system and unplug the power cord.

7) Touch the chassis of the host computer system to dissipate any static charge.

8) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

9) Touch the chassis of the PC to dissipate any built up static charge.

10) Securely install the JTAG board in an available PCI slot in the host computer.

11) Connect the JTAG pod to the host-pod cable. Connect the host-pod cable to the connector located on the end bracket of the JTAG PCI plug-in board.

Baseboard Installation

To install the baseboard:

12) Perform the board installation in an “ESD” or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

13) Shut down Windows and power-off the host system and unplug the power cord

14) Touch the chassis of the host computer system to dissipate any static charge.

15) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

16) Touch the chassis of the PC to dissipate any built up static charge.

17) Connect the 14-pin connector on the JTAG PCI pod to the DSP board JTAG connector (Non-DSP board users skip this step).

18) Securely install the baseboard into an available PCI slot in the host computer.

IMPORTANT: Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

A Few Considerations BEFORE Power-up

Double-check all connections before applying power. Ensure that the JTAG and baseboard cards seated correctly in the slot.

It cannot be overemphasized:
Double check your cabling **BEFORE** connecting to the baseboard. **DO NOT** hot plug the cables. Hot plugging cables can cause latch-up of components on the card and damage them irreparably. Be aware that the cables to analog inputs are an important part of keeping signals clean and noise-free. Shielded cables and differential inputs (where applicable) help to control and reduce noise.

After Power-up
After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the "TroubleShooting" section.

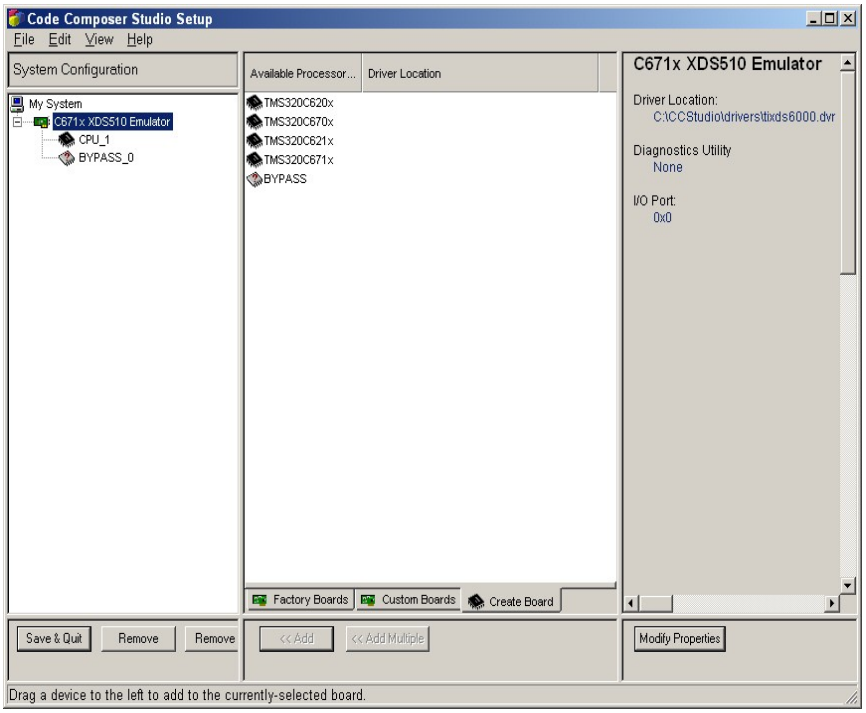
The next section is NOT used with the Non-DSP products. If the board you are installing is a Non-DSP product, the installation is complete.

Code Composer Studio Setup with II Jtag

To setup Code Composer Studio and activate the Innovative Integration-supplied CodeHammer JTAG board driver, the Code Composer Studio Setup Utility must be run. Since the Code Hammer debugger is XDS510- compatible, Code Composer Studio setup must be configured to use the XDS510 driver for the C6xxx. To do this:

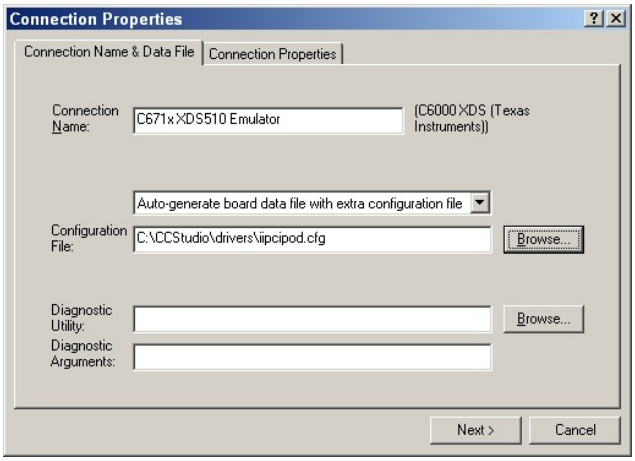
19) Launch the Code Composer Studio Setup Utility and remove the default simulator driver from the System Configuration. (right click the default simulator in the "System Configuration" pane and select "Remove")

20) Click the C6xxx XDS driver from the “Available Emulator Types” control within the setup utility and drag it into the “System Configuration” control.



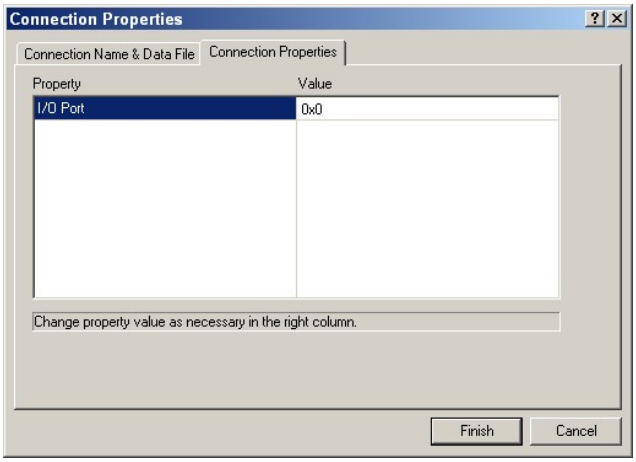
21) Once your emulator is added, a list of Available Processors is presented. Add the appropriate processors for your board as shown in the example. The example shows a set-up that is configured for an SBC6713e baseboard. The C671x emulator is selected as the baseboard uses the TMS320C6713 DSP.

22) Right-click on the C6xxx XDS emulator in the “System Configuration” Pane and select “Properties…” to invoke the **Connection Properties** Dialog for the driver.



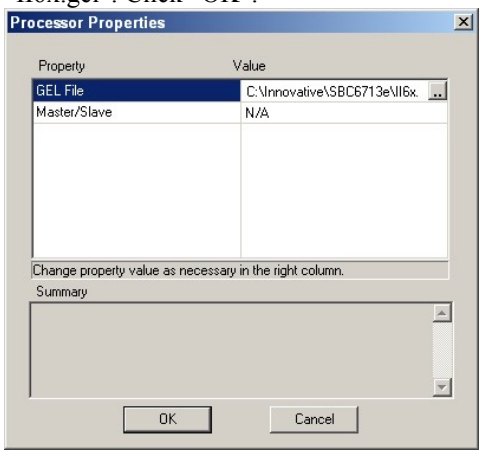
23) Under the “Connection Name & Data File” tab, the “**Connection Name**” edit box should match the emulator selected in the “**System Configuration**” Pane of the previous window. Change the “**Configuration File**” combo box to Auto-generate board data file with extra configuration file. Change the “**Configuration File**” edit box to "<drive>:\Cstudio\Drivers\IIPciPod.cfg". *<drive> is the letter for the drive onto which CCS is installed.*

24) Click the "**Connection Properties**" tab. Set the I/O port value for the driver to virtual device address "0x0" and click "**Finish**".



25) The main Code Composer Studio Setup window is now back in focus. The processor must now be configured. To do this: select the processor as shown in the "**System Configuration**" Pane (in our example "CPU_1"). Right click "CPU_1" and select "Properties…".

26) The "**Processor Properties**" screen will be presented. Click GEL File, click the ellipsis (…) and navigate to the Innovative Integration board install directory (typically C:\Innovative\BoardName) and select "II6x.gel". Click "OK".

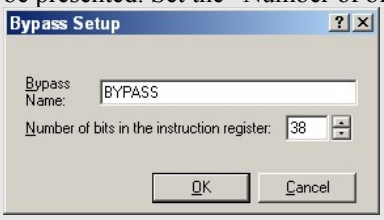


27) Click "**Save & Quit**" to save the configuration and exit the setup tool. You will then be prompted to launch **Code Composer Studio**.

Note: For multi-target boards such as the Quadia, one processor should be added for each device in the JTAG scan path.

Note: The SBC6713e has (2) DSPs, a C6713 and a DM642. Typically the DM642 should be set to "BYPASS" by selecting "BYPASS" from the "Available Emulator Types" control within the setup utility and drag it into the "System Configuration" control. Once this is done, the following screen will

be presented. Set the “Number of bits in the instruction register” to “38” and click “OK”



If you encounter difficulty launching CCS

28) Run the **JtagDiag.exe** utility (Start | All Programs | Innovative | Common Applets | JTAG Diagnostics) to reset the debugger interface.

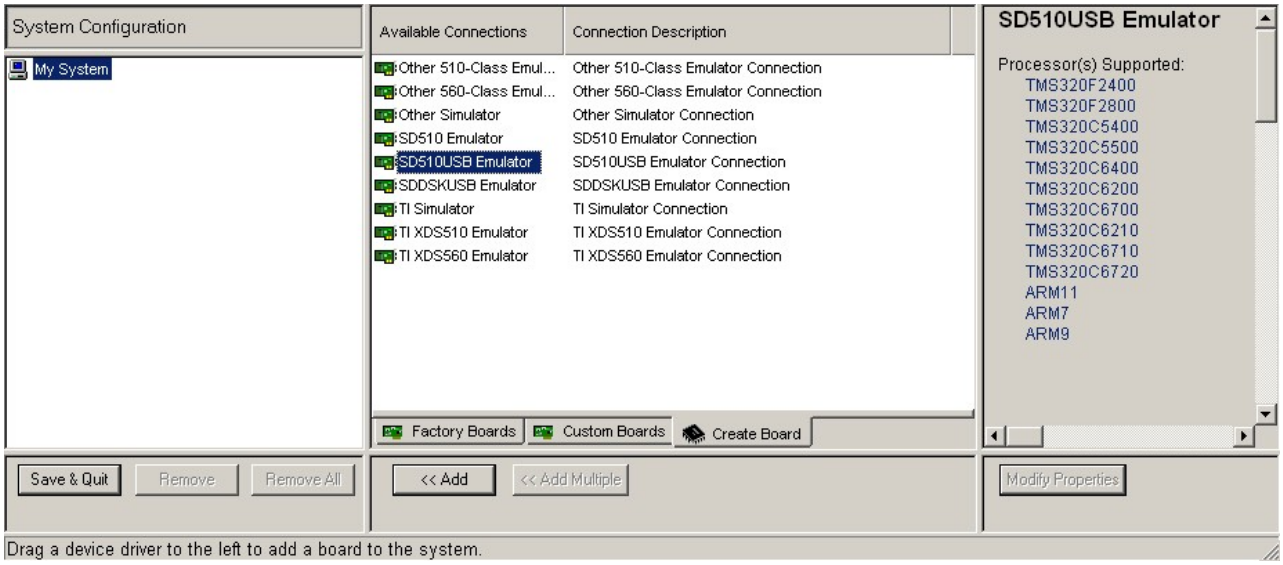
29) Run the board **Downloader utility** (Start | All Programs | Innovative | <Board Name> |<Applets> — Open the **Downloader Folder** and double click “**Downloader.exe**”) and press the Boot button (*Light Bulb icon*), to boot a default target application.

30) Restart Code Composer Studio.

Setting up for a single processor with Spectrum Digital USB Jtag.

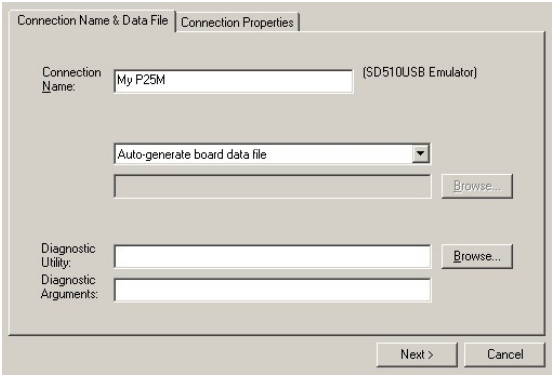
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



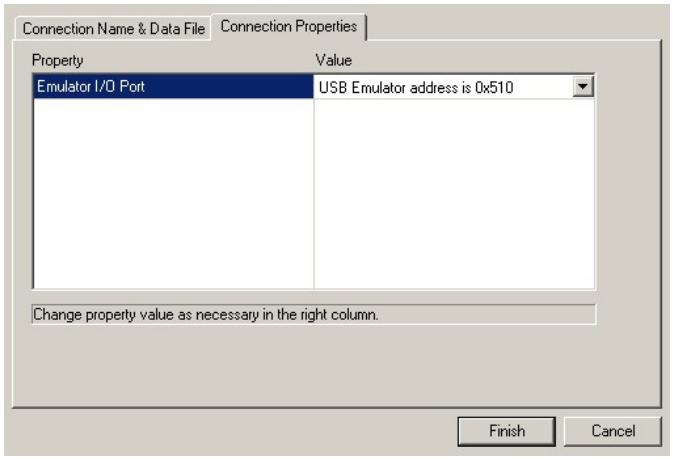
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



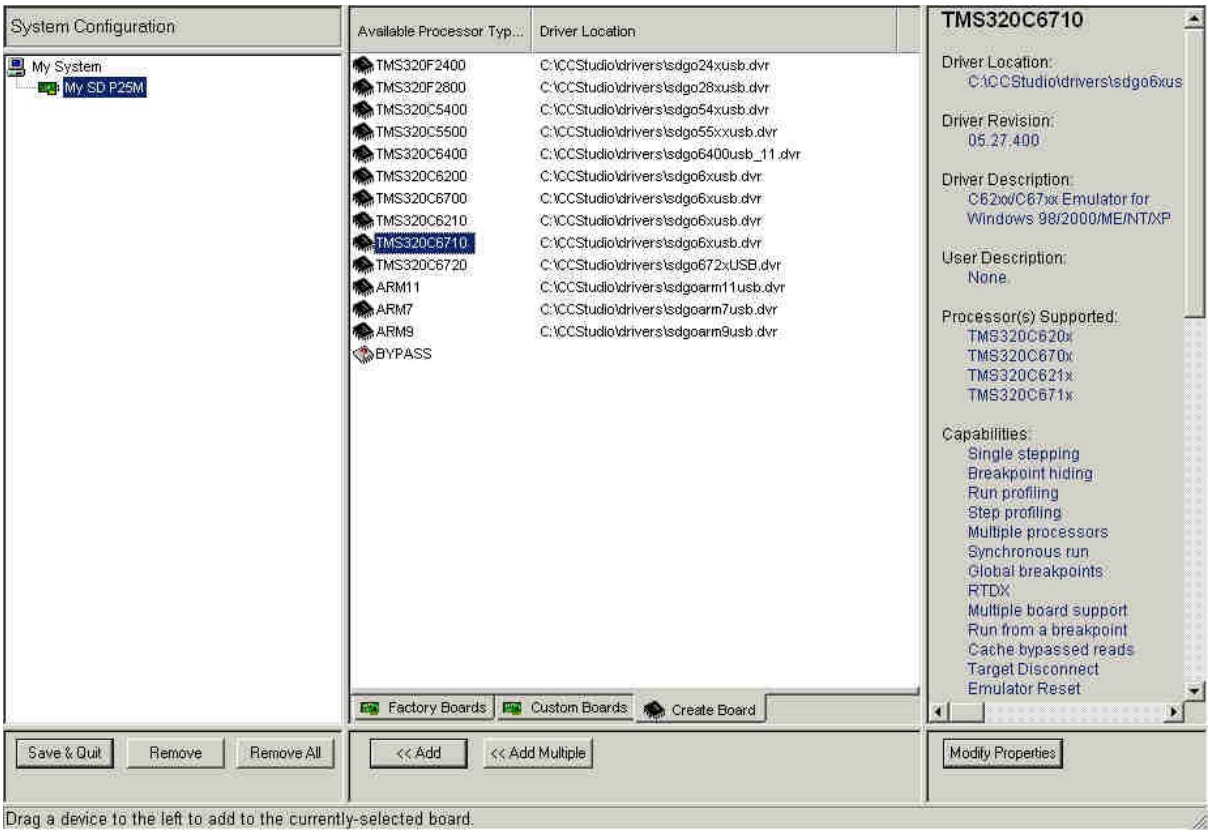
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

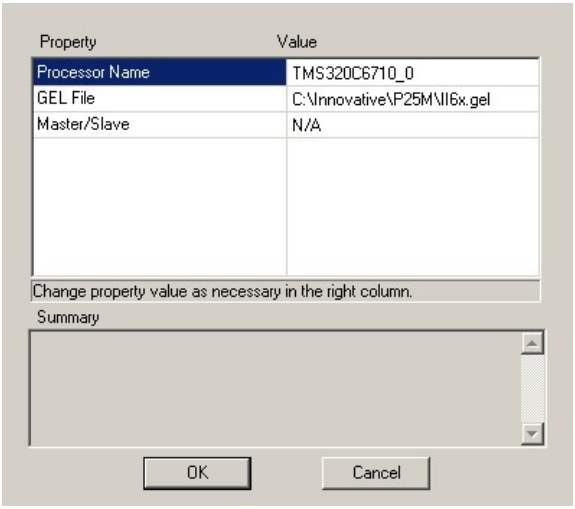


Now we add a processor

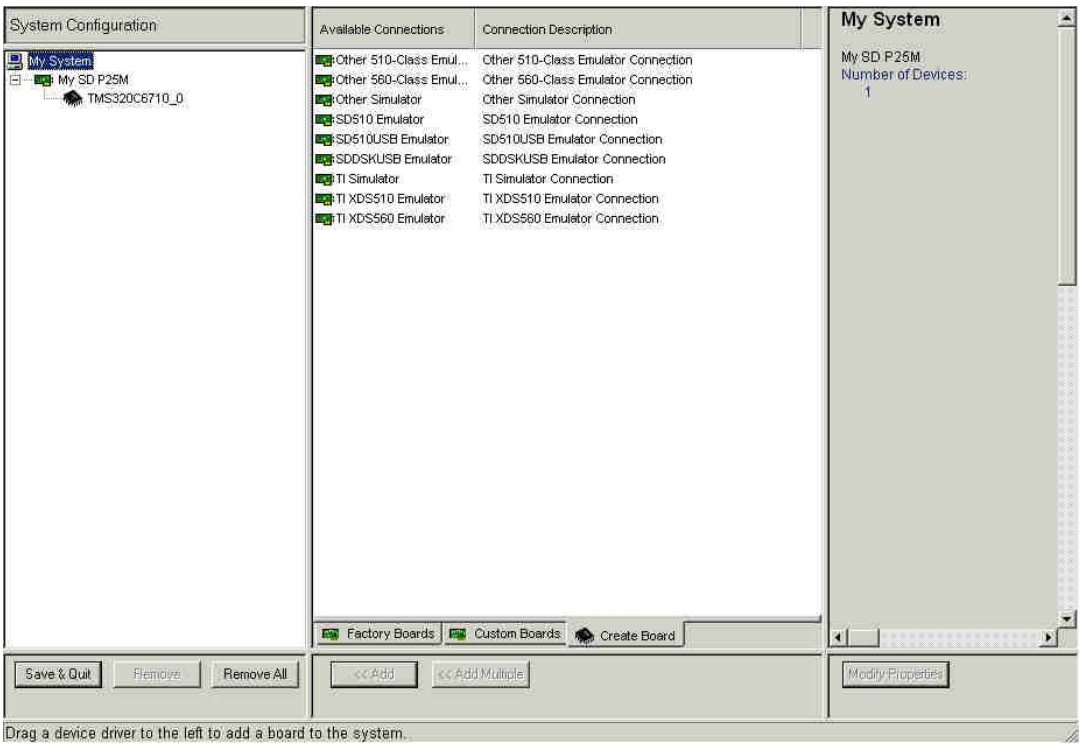
Each if the II boards have different processors so match up the closest one for your board.



Use the property sheet to find the Gel file from Innovative for your specific board.



Your system will look similar to this. Save the configuration and quit.



Setting up for Multi Processors with Spectrum Digital USB Jtag.

For the multi-processor setups use the following type setup. This includes the SBC6713e, Quadia, Q6x type Innovative boards.

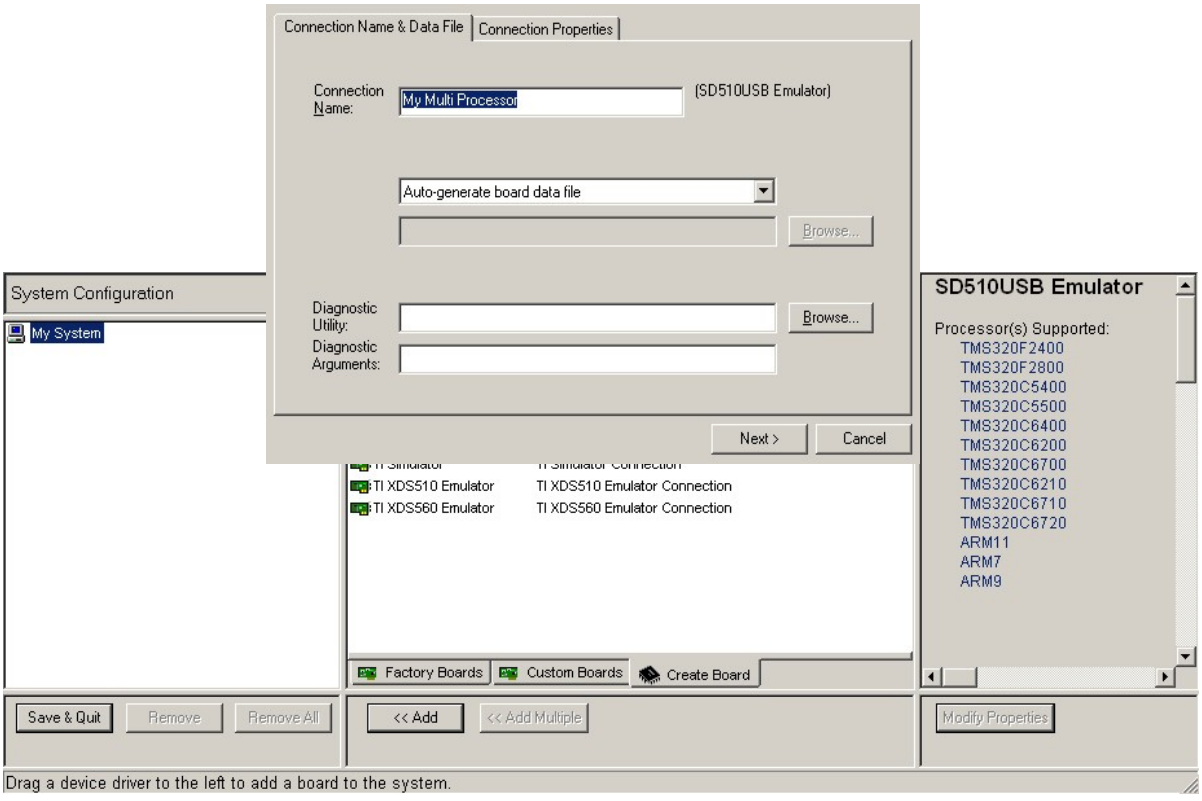
The SBC6713e board shown will be similar in setup with the other boards. The differences will be in the types of processors and the number added.

First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.

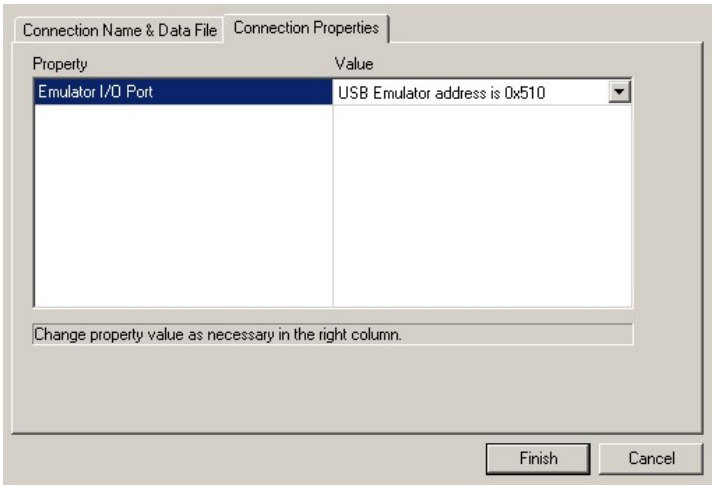
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



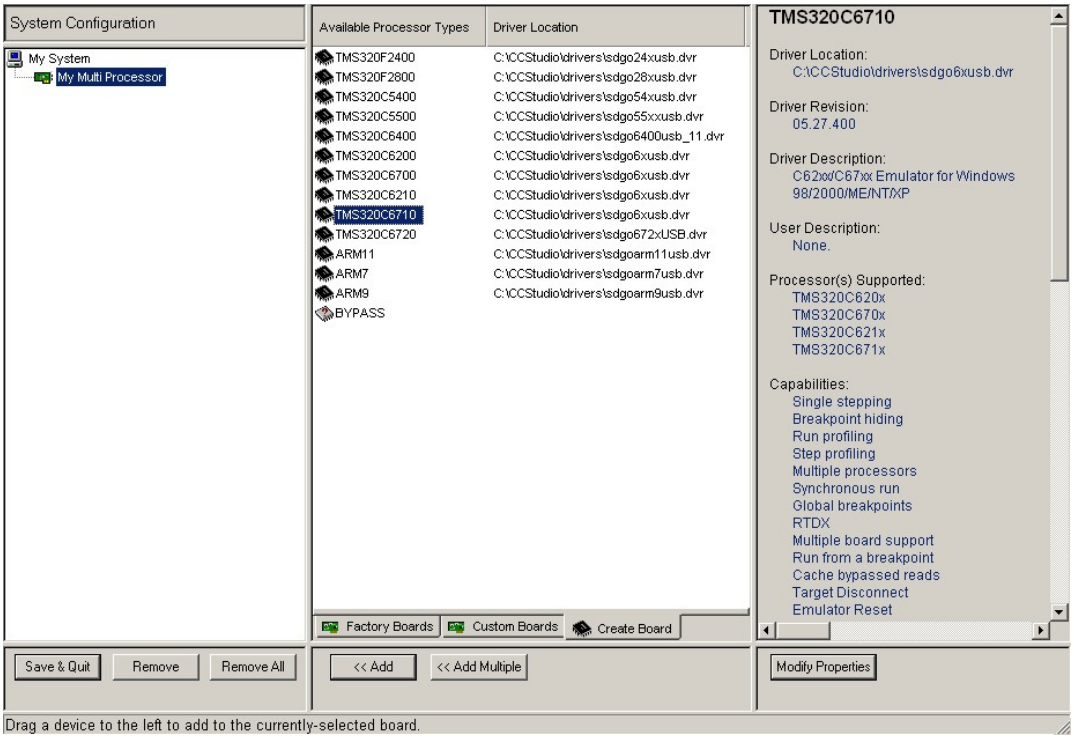
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

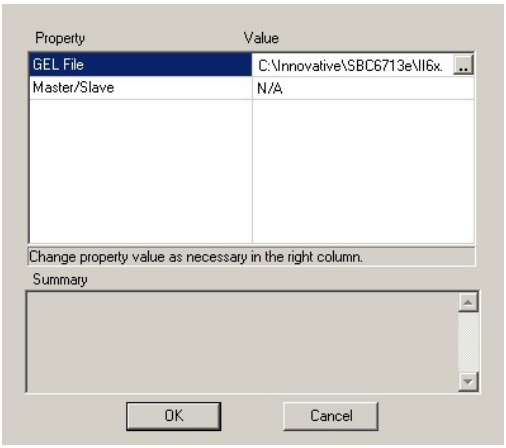


Now we add a processor

Each if the II boards have different processors so match up the closest one for your board.

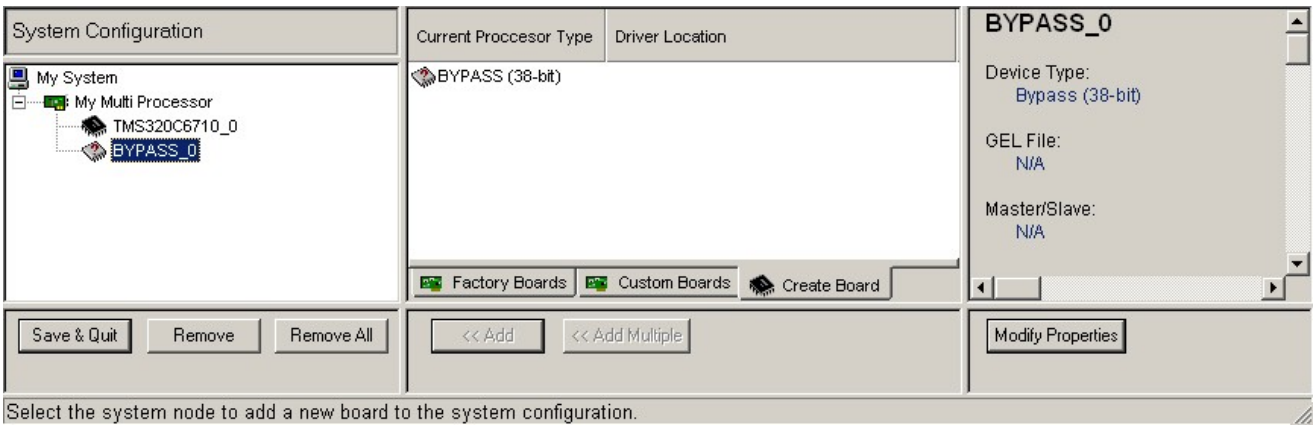


Use the property sheet to find the Gel file from Innovative for your specific board.



Close the processor and choose another processor. This will be a bypass for the DM642. Set the bypass for 38 bits. (For TMS6713 bypass use 42 bits on the first processor, the second processor will be a 64xx and the gel file from II for the DM642). For the Quadia use another C6400 type processor totaling 4 processors. All 4 will use the same GEL file from II.

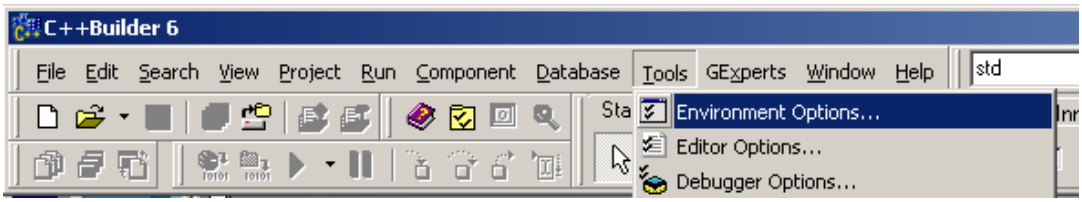
Your system will look similar to this. Save the configuration and quit.



Borland Builder Setup and Use

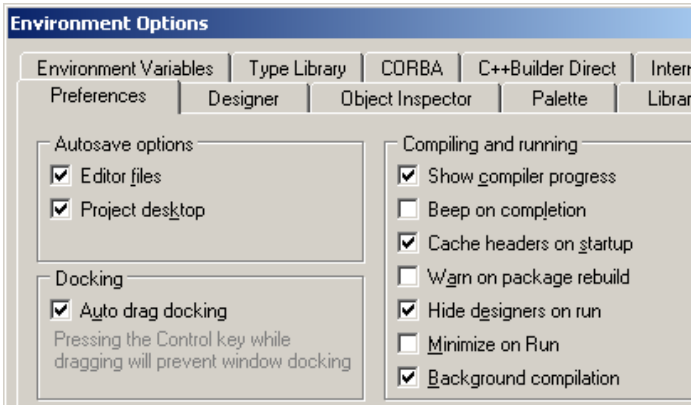
Following the normal installation of the Innovative Integration toolset components, numerous VCL components and C++ classes are automatically added to the BCB IDE. Additionally, Innovative recommends that the following IDE and project options be manually changed in order to insure simplified use and proper operation:

Automatic saving of project files and forms during debugging



31) Select Tools | Environment Options... from the main BCB toolbar.

32) This will invoke the **Environment Options** dialog:



33) Click the Preferences Tab

34) Check "**Editor files**" and "**Project desktop**" under "Autosave Options" so that project files are automatically saved each time a project is rebuilt and debugged.

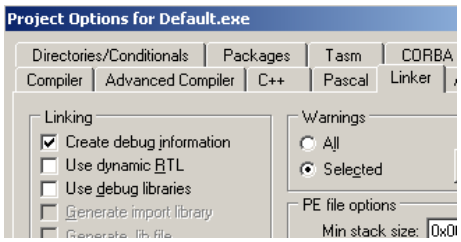
35) Click "OK"

Static-binding of built executables.

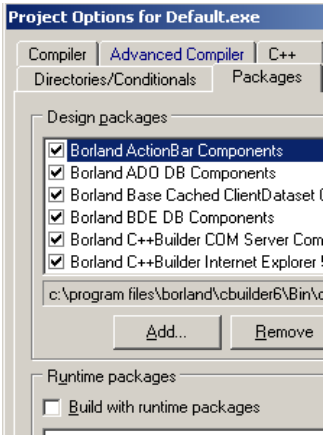
36) Click on Project | Options on the main BCB toolbar to invoke the **Project Options dialog**.

37) Click the **Linker** tab.

38) Uncheck the "Use Dynamic RTL" checkbox.



39) Next, click on the Packages tab and uncheck the "Build with runtime packages" checkbox.



These options insure that projects are built with minimal dependencies on external DLLs. See the FAQ "**What DLLs do I have to deploy with my newly created executable**" in the Troubleshooting chapter for details on which DLLs must be deployed with user-written executables.

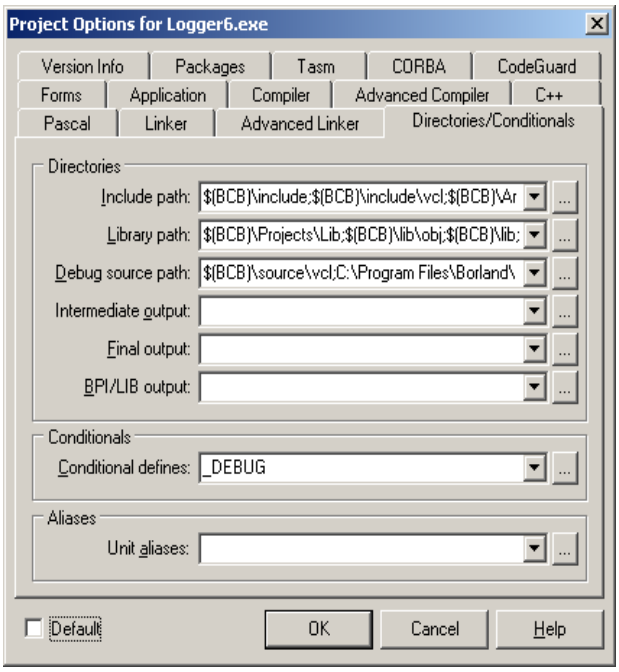
Appropriate library and include paths.

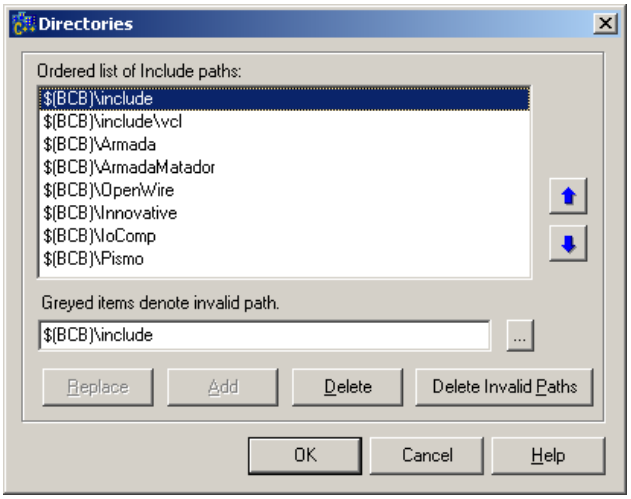
40) Click on the "Directories/Conditionals" tab.

41) Click the ellipses next to the Include Path edit box to invoke the Include Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

42) Next, click on the ellipses next to the Library Path edit box to invoke the Library Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

These changes insure that the standard Armada headers and object files are available to projects during compilation. Note that these paths may either be added to the default BCB project, by editing these options without first opening a specific project, or to specific projects after opening them. The advantage of the former is that the settings are automatically present on all subsequently-created projects.


Project Options for Logger6.exe
Version Info
Packages
Tasm
CORBA
CodeGuard
Forms
Application
Compiler
Advanced Compiler
C++
Pascal
Linker
Advanced Linker
Directories/Conditionals
Directories
Include path:
$(BCB)\include;$(BCB)\include\vcl;$(BCB)\Ar
Library path:
$(BCB)\Projects\Lib;$(BCB)\lib\obj;$(BCB)\lib;
Debug source path:
$(BCB)\source\vcl;C:\Program Files\Borland\
Intermediate output:
Final output:
BPI/LIB output:
Conditionals
Conditional defines:
_DEBUG
Aliases
Unit aliases:
Default
OK
Cancel
Help


Directories
Ordered list of Include paths:
$(BCB)\include
$(BCB)\include\vcl
$(BCB)\Armada
$(BCB)\ArmadaMatador
$(BCB)\OpenWire
$(BCB)\Innovative
$(BCB)\IoComp
$(BCB)\Pismo
Greyed items denote invalid path.
$(BCB)\include
...
Replace
Add
Delete
Delete Invalid Paths
OK
Cancel
Help

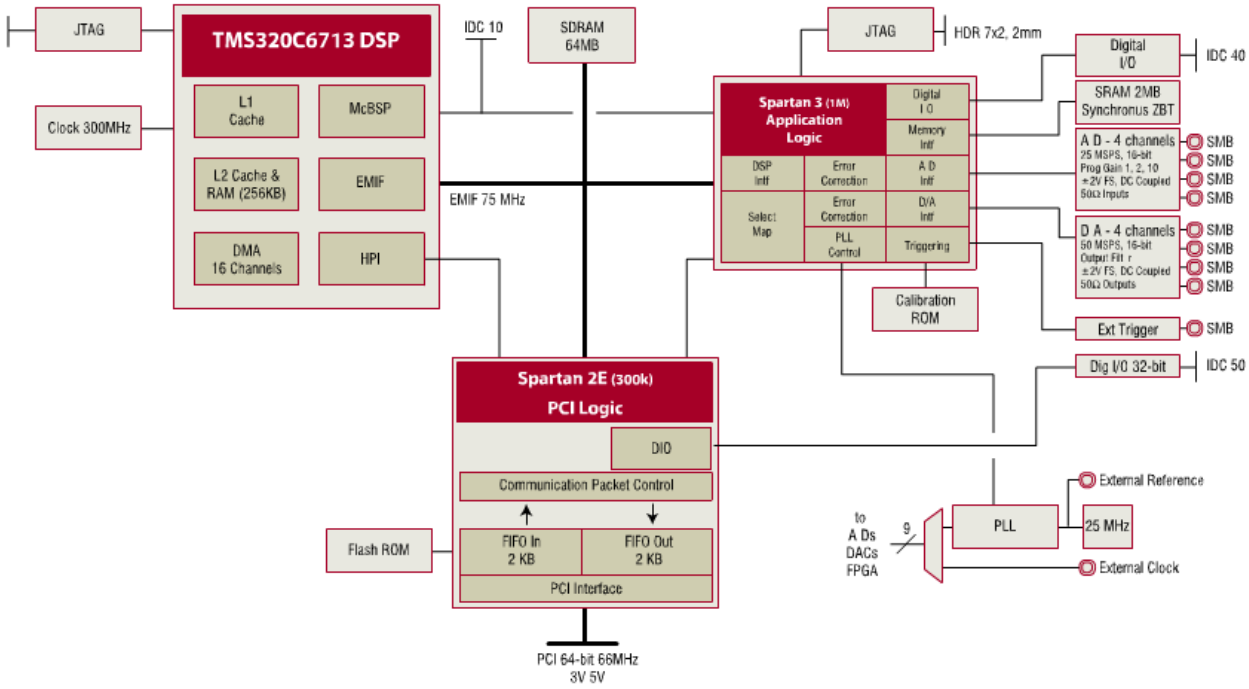
About the Baseboard

DSP Baseboard Hardware Features

The P25M baseboard features a Texas Instruments TMS320C6713 digital signal processor with 128 M bytes of SDRAM memory and a Xilinx Spartan3 1M FPGA with 2MB local SRAM.

To complement this signal processing , the P25M has four channels of 25 MSPS, 16-bit A/D and four channels of 50 MSPS, 16-bit D/A and two digital IO ports totaling 64 bits of IO. The combination of DSP and FPGA signal processing with analog and digitla IO makes the P25M ideal for a variety of real-tie signal processing applications including servo applications, data acquisition, stimulus- response measurements and many others.

The P25M has a number of other features for system integration such as a high performance PCI interface capable of burst rates up to 512 MB/s, precision low-noise sample clock generation, external clock and trigger inputs, and on-card power supplies. Each of these features makes the P25M easy to integrate into system designs to add high speed, real-time signal processing and analog IO.



Digital Signal Processor

The P25M baseboard uses the Texas Instruments TMS320C6713 DSP, operating at 300 MHz. This DSP is a 32-bit floating point device with a 32-bit external memory bus. The DSP interfaces to the memory and peripherals on the baseboard through its external memory interface (EMIF), which has programmable definitions for the memory interface timing.

DSP External Memory

The P25M baseboard provides 128M bytes of SDRAM memory mapped to the '6713 DSP CE2 memory space. This memory provides space for program and data storage as well as storage space for data acquired/generated by the baseboard analog hardware. This memory is programmed to operate at 75 MHz regardless of the DSP core clock rate. The initialization of all external memory spaces are defined within the file HdwLib\IIInit.cpp and include the correct parameters for the type of SDRAM used on the baseboard, including refresh timing, as well as timings for all sync and async peripherals and should not be modified.

DSP Initialization

For proper operation of the external peripheral on the baseboard, the external memory interface control registers must be configured prior to use of the external memory interface. Applications built under the Pismo Toolset libraries will automatically initialize the registers appropriately (using code within HdwLib\IIInit.cpp). For those customers who need to initialize the registers manually, please refer to the EMIF register initialization values within the IIInit.cpp source file to obtain the required register values. Please note that the initialization is order sensitive and should be performed in the order given in the table below.

Table 1. 'C6713 DSP EMIF Control Register Initialization Values

Register Name	Address	Value	Use
GBLCTL	0x01800000	0x00003078	
CE0CTL	0x01800008	0x10910321	Asynchronous devices
CE1CTL	0x01800004	0x00000040	PCI Interface FIFOs
CE2CTL	0x01800010	0x00000030	SDRAM
CE3CTL	0x01800014	0x00000040	Analog IO FIFOs
SDTIM	0x0180001C	0x00000350	
SDEXT	0x01800020	0x000544a7	
SDCTL	0x01800018	0x6B338000	

During the development process, code may be downloaded to the baseboard using a JTAG debugger or via the PCI bus interface. After development is complete, the debugged application image may be downloaded from within a Host application using standard functions within the Malibu libraries.

DSP JTAG Debugger Support

Standard TMS320 family JTAG debugger operation is supported by each baseboard. An external debugger connector is supplied that allows use of industry standard JTAG debugger hardware from Innovative, Texas Instruments, and other third party suppliers. The DSP is the only device in the DSP scan path; a separate scan path is provided for the Xilinx devices. Software for JTAG debugging and code development is TI Code Composer Studio.

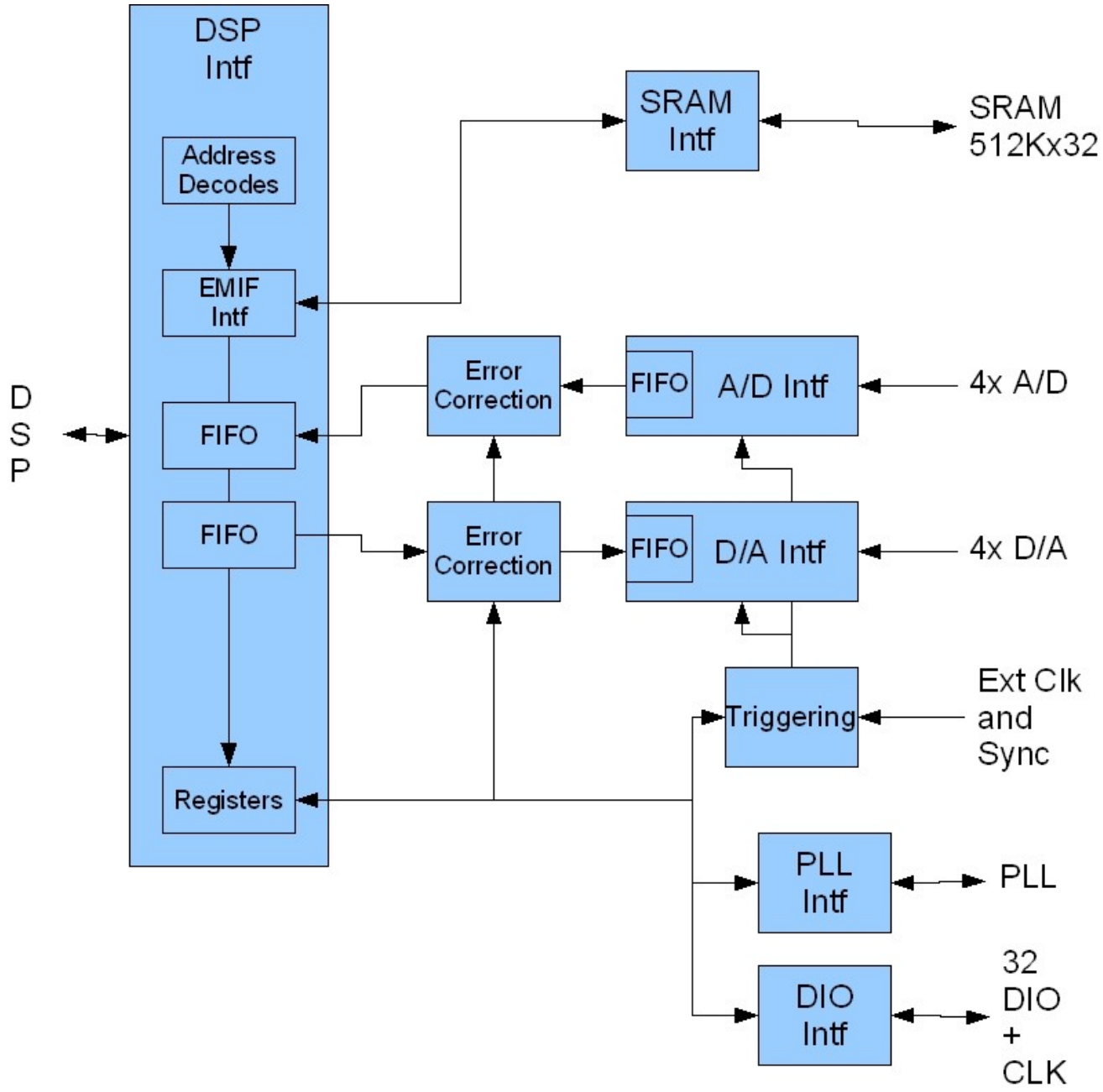
FPGA Core

The P25M has a 1M gate Xilinx Spartan3 FPGA with 2MB of local SRAM that performs many IO functions and may be reprogrammed to add signal processing or unique features to the P25M. The FPGA directly connects to the analog and digital IO, as well as the DSP, allowing access to the high speed, real-time data streams between the DSP and the IO.

FPGA Device	Density
Xilinx XC3S1000-4FG456C	1M gates

A standard logic image, referred to as the FrameWork Logic for P25M, provides support for for using the P25M analog and digital IO. The logic features include IO interfaces, error correction, data buffering, triggering and DSP interface control amongst others. The FrameWork Logic is used in many P25M applications since it meets many application requirements and is used as the basis for custom logic designs.

The following block diagram of the logic shows the basic connections implemented in the FrameWork Logic design. The FPGA provides the interface between the DSP and the IO devices through memory-mapped registers and FIFOs. Analog support functions such as error correction and triggering are configured by the DSP during initialization, allowing the logic to capture real-time data and perform signal processing independently of the DSP. Data streams between the DSP and analog using FIFO interfaces are provided using interrupt-driven DMA or CPU accesses.



FPGA Memory

The FPGA core has a private 2MB SRAM device configured as 512Kx32. This device is used in custom logic applications as a data buffer for the FPGA. The memory interface is fully controlled from the FPGA and is nominally clocked at 75MHz.

FPGA Clocks

The application logic is clocked by the DSP EMIF interface at 75 MHz and by a sample clock input. The sample clock input is used for the analog IO capture and triggering. Custom applications can also bring in external clocks or use the FPGA internal clock resources to generate other clocks.

Custom Logic Applications

Complete programming information for the P25M FPGA is provided in the *FrameWork Logic User Guide* and *FrameWork Logic MATLAB User Guide* . These manuals provide detailed information on reprogramming the FPGA using VHDL and MATLAB toolsets for custom applications.

Loading the Application Logic

The P25M application FPGA logic must be loaded as part of the initialization after power up. The logic is volatile and is NOT preserved at power down.

The application logic is usually loaded using either the LogicDownloader applet, or as part of the host control program using the Malibu support software libraries. The logic may dynamically loaded at any time, allowing applications to have several logic images that customize the card functions according to the requirements. The logic downloads use the Xilinx SelectMAP interface accessed via the PCI bus interface. The host is responsible for the logic download, not the DSP.

During logic development, the logic can also be downloaded over the JTAG interface. The P25M has a separate JTAG interface for the Xilinx devices that connects to a standard Xilinx Platform USB or Parallel IV cable via connector JP5 (see appendix for pinout). The scan order is as follows for the Xilinx JTAG chain. The FLASH and PCI interface FPGA should be placed in bypass mode when using this mode.

Device Number	Device Function	Device
0	PCI FLASH EEPROM	XCF04S
1	PCI Interface FPGA	XC2S300E
2	Application FPGA	XC3S1000

The PCI interface FPGA has an on-card serial FLASH device that is automatically loaded at power-up. The FLASH device may be occasionally updated when new logic is released by Innovative. The applet *EEPROM* is used for this reprogramming.

FPGA Initialization

The FPGA must be initialized after the image is loaded and the DSP is initialized. An access to DSP CE0 (0x80000000) releases the FPGA clock DCM from reset making the FPGA ready for use.

The Pismo Class Library

Innovative Integration's **Pismo** is a software class library allows the developer to fully exploit the advanced hardware features of the Innovative DSP product lines and to reap all the benefits from Texas Instrument's DSP/BIOS Operating system. Every board peripheral has been carefully integrated into the OS and its functionality encapsulated in a device driver that can readily be controlled within DSP/ BIOS applications including PCI interface, analog I/O, external bus and memory, serial ports and other I/O devices.

Pismo provides extensive C++ class support for:

- Dynamic creation and runtime control of tasks
- Simplified management of and access to all TI Chip Support Library (CSL) and DSP/BIOS API functions including: Semaphores, Mutexes, Mailboxes, Timers, Edma, Qdma, Atoms, McBsp, Timebases, Counters, etc.
- Foundation (base) classes for DMA-driven device driver development
- Templatized queues
- Partial standard-template library functionality via STLPort

For example, the code fragment below uses the Pismo IntBuffer class to initialize a QDMA (quick DMA) to perform a memory-to-memory move of a constant value (0) into a 4096-word buffer (at Src), then to copy the source buffer (Src) to the destination buffer (Dst):

```
// Create a source buffer of 0x1000 integers
IIBuffer Src(0x1000);
// Initialize the source buffer with zeros
Src.Set(0);
// Create a destination buffer of 0x1000 integers
IIBuffer Dst(0x1000);
Dst.Copy(Src);
```

Simple To Use

In the same way, peripheral-specific class libraries dramatically simplify access to board-specific peripheral features. For example, the code fragment below illustrates real-time processing and display of analog input signals running on the P25M DSP board equipped with an Omnibus module within a separate thread of execution:

```
//-------------------------------------------------------------------------
//  LoopThread() --  Capture snapshots of A/D input
//-------------------------------------------------------------------------

class LoopThread : public Thread
{
public:
    LoopThread(IIPriority priority)
        : Thread(priority), FCount(0), Cursor(0), Requested(false)  { }

    int             Count()
        {  return FCount;  }
    void            Resize(int size)
```

```
        {  CaptureEvents=size;  Snaps.Resize(size);  }
    IntBuffer &     Acquire()
        {
        Cursor = 0;
        Requested = true;
        Available.Acquire();

        return Snaps;
        }

protected:
    // Fields
    volatile int    FCount;

    // Data
    bool            Requested;
    Semaphore       Available;
    IntBuffer       Snaps;
    int             Cursor;
    int             CaptureEvents;

    // Methods
    void Execute()
        {
        // echo input to output
        while(!Terminated())
            {
            AIn.Get();
            ++FCount;

            //
            // If main thread wants a block, copy it to him
            if (!Requested)
                continue;

            int Residual = Snaps.Ints()-Cursor;
            int Chunk = std::min(Residual, AIn.Buffer().Ints());
            Snaps.Copy(AIn.Buffer(), Cursor, Chunk);
            Cursor += Chunk;

            if (Cursor == CaptureEvents)
                {
                Requested = false;
                Available.Release();
                }
            }
        }
};
LoopThread Loop(tpHigher);
```

Not Just for C++ Experts

Note that even if you're not a C++ maven, the code is quite clear and understandable. In fact, one of the benefits of using C++ is that while it helps to mitigate and manage complexity to support creation of larger, more sophisticated applications, it is often simply used as a "better" dialect of the C language. C++ is essentially a superset of C. As such, you may freely intermix calls to legacy 'C' functions, newly-written C functions, Assembler functions and C++ functions (called methods) within C++ programs. You need not fully understand all of the enhanced capabilities and features of C++ in order to fully exploit the features of the class libraries provided in Pismo.

Analog I/O Streams

The Analog I/O is, for most applications, the most important feature of the P25M baseboard. Most of the peripherals on the hardware are related to Analog I/O. Most of the configuration options are related to Analog I/O. It is the part that causes the most problems in development.

To maximize the chances for success, the Pismo library provides a set of classes that hide all of the details of data acquisition. From the application level, the user simply processes buffers of data. The details of hardware and software management are isolated from the application.

Stream Objects and Device Drivers

Data I/O in DSP/BIOS is accessed and controlled via custom device drivers. Access to the device driver is controlled by a Stream class. These drivers are dynamically installed by the Stream when needed by the user application. From the point of view of the application the stream control class provides all of the user interface function needed to configure and operate the I/O operation.

Table 2. Device Driver and Stream Classes

Device Driver Class	Stream Class	Description
AnalogInputDriver	AnalogInStream	Streamed Input from an analog source. Continuous data flow with buffering. Data flow stops only via trigger control.
AnalogOutputDriver	AnalogOutStream	Streamed Output to an analog source. Continuous data flow with buffering. Data flow stops only via trigger control.
CaptureInputDriver	CaptureInStream	Burst Input from an analog source. Data flow is discontinuous, filling each buffer requested and stopping.
ServoBase	ServoIntf	Continuous, low-latency analog capture and playback suitable for performance servo control applications. Event-at-a-time application data processing.

Hardware Isolation and Independence
The Analog and the Capture driver allow a single Stream to be used with different analog hardware. In the P25M, for example, Omnibus modules allow a wide variety of analog choices on a single baseboard. Each of the Stream classes can be "attached" to a particular module and will automatically configure itself to use that hardware.

The Servo driver provides low-latency, interrupt-driven data processing, suitable for real-time control applications, albeit at the expense of high CPU usage. It is currently implemented only for the Servo16 module.

The FpdpInStream and FpdpOutStream drivers provide communications with external Front Panel Data Port devices.

Stream I/O Types
There are two distinct categories of Streams implemented within Pismo - Continuous and Burst.

Continuous Streams use the model that the input or output is a continual process, whether periodic or not. Thus in order to avoid data loss when the application is momentarily busy, internal buffering is provided so that the hardware may operate for extended periods without software intervention. This means latency must be increased: data may be 'in the queue' for some time until additional data forces it out to the application. Of course, if the application does not process the data as fast as it arrives data will eventually be lost.

Burst Streams use a different model. Here data movement is 'on demand' instead of asynchronous. If no request for action by the application is received, the Stream is idle. This type of Stream is more common for non-Analog I/O such as the FifoPort or PCI busmastering, but the CaptureInStream implements a burst type I/O model on the analog hardware.

Stream Buffer Model

Each Stream uses data buffer class objects to pass data between its hardware and the application. These buffers are all the same size. Passing data between Streams is simple if the buffers are chosen to be the same size:

```
Ain.Get();
Aout.Put(Ain.Buffer());
```

Buffer transfers are efficient because the data buffers are not copied at any time during the transfer process. By default, Streams allocates three buffers, two internal and one swap buffer. If desired, the number of internal buffers in the pool may be modified prior to opening the Stream by assigning a new value using the BufferCount method.

```
// Instantiate the analog stream objects
AnalogOutStream Aout;
Aout.BufferCount(5);
AnalogInStream Ain;
Ain.BufferCount(5);
```

This code forces the Stream to allocate five internal buffers.

The size of data buffers may be specified explicitly using the Stream::Events method. This latter method sizes the buffers such that they can contain the at least the specified number of acquisition "events", where an event is defined as one sample from all enabled A/D or D/A channels. This simplifies most buffer processing algorithms since all buffers are guaranteed to contain an integral number of samples from all enabled channels.

The product of the buffer size and the number of buffers gives the load-carrying capacity of the system. For example, the originally allocated three buffers per stream, each sized at 0x1000 bytes, running at 44.1 kHz equates to a load carrying capacity of

```
(0x1000 bytes/buffer) x (3 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 139 mS
```

Whereas in the second example, with six buffers per driver pool

```
(0x1000 bytes/buffer) x (6 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 278 mS
```

Data integrity can thus be preserved at the expense of additional memory utilization.

Burst Streams place data into the buffer provided. No buffering is used, and data acquisition is halted when the provided data buffer is filled.

Stream Internals
The DSP CPU used on the P25M is powerful and fast, yet the Stream classes improve performance even further by drastically reducing CPU use for data movement. The available DMA channels in the C6000 DSPs are fully exploited to do movement to and from hardware to memory so that hardware interrupt rates rarely exceed 1KHz! The net effect is that virtually all of the bandwidth of the CPU is available for application processing, without requiring any application DMA programming.

Multitasking Friendly
The Stream classes support efficient cooperation in multitasking applications. Any function that requires a delay to complete. will block using DSP/BIOS functions that release other OS threads for efficient utilization of the processor.

Using Analog Streams in an Application

The AnalogInStream, AnalogOutStream, and CaptureInStream all allow fast data movement between the application and the hardware in different modes. Once associated with a hardware device, they allow all configuration and control of the session to take place through the methods of the Stream.

Every Stream must consider these questions to make a functioning application:

- Which stream to use (Input vs. Output or Continuous vs. Burst).
- Which hardware to use, and in which hardware mode.
- Which clock source to use, and with what parameters.
- Which triggering mode to use, and with what parameters.

Once these are taken care of, using the Stream to perform the Analog I/O is a simple matter. Consider the code fragment below which illustrates all of the steps necessary to fully initialize and stream a stream a continuous 1 kHz sine wave to the analog outputs present on a SD16 Omnibus module attached to an P25M baseboard SD16 module at 50 kHz:

```
using namespace II;

//--------------------------------------------------------------------------
//  IIMain() --  Illustrate Omnibus Output
//--------------------------------------------------------------------------

AnalogOutStream  Aout(Omnibus::mSite0);

void IIMain()
{
        // ...Load Module onto site 0.
LoadModule(Omnibus::mSite0, Omnibus::mtSD16);
        //
//  ...Use default clock (DDS)
        //  ...Set Clock Rate
ClockRateUIPtr(Aout.Clock())->Rate(50000);
        //
        //  ...Output on all channels
Aout.Channels()->EnableAll();
        //...Size the buffers
        Aout.Events(5000);
```

```
Aout.BufferCount(BuffersPerSec/3);

        // Open the streams
        Aout.Open();

        // Stream loop...
        bool run = true;
        while ( run )
                {
                Aout.Generate();
                Aout.Put();
                }

                // Terminate streaming
        Aout.Close();
}
```

Examining the above code, you can see the application uses an AnalogOutStream, since it is an output program. The next interesting line is the call to LoadModule(). This informs the system of which module is 'plugged in' on Omnibus::mSite0, where the stream is also attached. With the attachment of the module to the stream, the stream object can configure hardware, clock, and trigger settings.

The next several lines configure the Stream clock rate, the channel configuration, and the stream buffer count and buffer size. Then comes the Stream::Open method which activates the device driver in DSP/ BIOS. Afterwards, the Stream::Control method may be used to perform any necessary device-specific initialization and/or control functions.

Data flow begins with the calls to Generate() and Put(). Generate() uses signal generator classes to write a signal pattern into the buffer. Put() enqueues the data for output. Data output will not actually begin until the buffer queue is essentially filled with data. This avoids under-runs of the output. For input Streams, the Get() method starts streaming at the first call. These buffer methods should be repeated to keep the streams flowing. After use of a device is complete, it is closed using the Stream::Close() method.

Note that in the above example, the type of module used matters very little in the finished application. Simply by changing a single constant this code can be rebuilt to work on any module that supports Analog output. The module specific details are handled by the Pismo library internally.

Selecting the Stream Object

Each Stream object is used to manage input or output on a single Omnibus module. Multiple module applications need to use separate instances of the stream for each module site. The Stream object is associated with a module site by the constructor, allowing access to the hardware for configuration:

```
    AnalogInStream  Ain(Omnibus::mSite0);
```

The AnalogInStream provides continuous streaming input. All data is delivered to the ring of internal buffers and from there to the application.

AnalogOutStream allows continuous streaming output to an output device. The application must deliver data as fast as it is consumed to avoid buffer underruns.

The CaptureInStream is a new type of driver that is used to emulate manual capturing of data to a buffer. In this mode, the analog hardware is inactive until a buffer is presented for filling. When this occurs, an acquisition is started that will fill the presented buffer, after which data taking is stopped. The process repeats for each buffer presented.

In the capture mode, no processing is taking place when data is not requested. This is a major difference from AnalogInStream. Also, if the buffer is sized to be smaller than the analog hardware's own FIFO or storage, each buffer will be a snapshot dump of the FIFO contents. This makes capture useful for snapshots of very high rate analog input, faster than the module can be read. There is no way to take continuous data sets larger than a single buffer in capture mode. There will be a gap between any two captures. AnalogInStream should be used for continuous applications.

Selecting and Configuring Hardware

The P25M supports Omnibus Modules, allowing multiple hardware configurations on the same baseboard. Modules are attached to an Omnibus site by a call to LoadModule().

```
// ...Load Module onto site 0.
LoadModule(Omnibus::mSite0, Omnibus::mtSD16);
```

Once a module is loaded, it can be accessed by the stream object's Module() method. This returns a generic pointer that can be converted to an exact module pointer to allow its methods to be called. The functions that do this are called Module Conversion Functions.

Module Conversion Functions are provided for all supported Omnibus modules. See the online help for the module class for a description of these functions. An example of the use of these functions:

```
SD16 * sd16 = SD16Ptr(AIn.Module());  // Gives valid SD16 object
A4D4 * a4d4 = A4D4Ptr(AIn.Module());  // Returns 0 -- not an A4D4!
```

Note that a module conversion function will fail if the conversion can not take place.

Selecting and Configuring Clocks

On attachment of the stream to a module, the Clock configuration system becomes active. It consists of the following methods:

Table 3. Stream Object Clock Methods

Method	Description
IsClockSourceSupported()	Returns True if a clock source is allowed on the hardware.
SetClockSource()	Change clock source to the selected source.
Clock()	Returns an interface object that allows the configuration of the Clock source.

The User Interface object returned by the Clock() method is used to configure the selected clock source. The possible ways a clock can be configured depends strongly on the type of source. For example, an internal clock such as the DDS can have the clock frequency programmed. An external clock can not. Rather than provide a complicated set of functions, many of which may not work for a clock source, we instead separate each distinct part of the User Interface into separate interface classes. The interface object for a clock source may support none, or one, or any number of all the possible UI interface classes. An

interface can be accessed by the conversion function for each of the UI interface classes. If an interface is not supported, the conversion function returns a null pointer.

The following code sample shows the use of conversion functions and multiple interface classes. The DDS clock source is in use. It supports both the ClockRateUI interface, which allows changing the clock frequency, and the ClockSyncUI interface, which allows configuration of the SyncLink/ClockLink master hardware to drive the DDS clock signal off the baseboard for use as a source on another board.

```
//  ...Set Clock Rate (allowed on DDS)
ClockRateUIPtr(AIn.Clock())->Rate(50000);
ClcokSyncUIPtr(AIn.Clock())->SyncLinkChannel(scSyncLink0);
```

Each line can be read from the inside out. AIn.Clock() returns the current clock UI object. This object is input into the clock conversion function ClockRateUIPtr() and converted into the ClockRateUI interface. Finally, the Rate() method of this class is called to set the rate of the clock to 50,000 Hz.

Selecting and Configuring Triggers

The Analog Stream objects allow the user to configure the triggering method used during the run. Triggering features are divided into four parts:

- Pretriggering - Handling data before the start trigger.
- Start Trigger - How to start data taking.
- Stop Trigger - How to stop data taking.
- Retriggering - How to handle interval before next start trigger

Selecting Pretriggering Modes

The pretrigger control for a stream consists of the following methods:

Table 4. Stream object Pretrigger Methods

Method	Description
IsPretriggerTypeSupported()	Returns True if the pretrigger mode is allowed on the hardware.
SetPretriggerType()	Change pretrigger to the selected mode.
Pretrigger()	Returns the pretrigger interface for the mode.

Table 5. Stream object Start Trigger Methods

Method	Description
IsStartTriggerTypeSupported()	Returns True if the start trigger mode is allowed on the hardware.
SetStartTriggerType()	Change start trigger to the selected mode.
StartTrigger()	Returns the start trigger interface for the mode.

Table 6. Stream object Stop Trigger Methods

Method	Description
IsStopTriggerTypeSupported()	Returns True if the stop trigger mode is allowed on the hardware.
SetStopTriggerType()	Change stop trigger to the selected mode.
StopTrigger()	Returns the stop trigger interface for the mode.

Table 7. Stream object Retrigger Methods

Method	Description
IsRetriggerTypeSupported()	Returns True if the retrigger mode is allowed on the hardware.
SetRetriggerType()	Change retrigger to the selected mode.
Retrigger()	Returns the retrigger interface for the mode.

Trigger configuration presents the same problem as the clock configuration, except more so. Triggering consists of four parts, each of which can be independently set. In addition, there are far more ways of defining triggers than there are for defining parts of a timer. For example, the start of data flow on Omnibus modules can be based off of an external digital signal, the value of the data on an input channel, by software command, or be automatic. Pretrigger and Stop trigger options are also numerous. A class that had methods for all these features would be large, complex, and would usually have most of its functions inoperative without giving the application any feedback.

Trigger configuration is also complicated by Module differences. Even the same type of trigger can differ in on different modules. For example, an external start trigger may allow the condition of the input signal (edge or level triggering, positive or negative polarity) be changed. Another module might not support changing these features, being always positive edge triggered. Another example is that threshold triggering might be able to be triggered of any selected channel, or it might be restricted to a predetermined channel.

The triggering configuration system uses trigger interface objects to allow configuration of the triggering modes. There are four access functions (Pretrigger(), StartTrigger(), StopTrigger(), and Retrigger()) giving interface objects for use. Each of these objects supports none, or one, or any number of Trigger UI interface classes for configuration, each of which can be exposed by a conversion function for the class. If a UI interface is not supported, the conversion function returns a null pointer.

The following code sample shows the use of trigger conversion functions and multiple interface classes. The AD40 module is in use. It supports several different Pretrigger and StartTrigger modes. In the example we wish to use Counted Pretriggering and Threshold Start Triggering.

```
//
//  This example for an AD40 uses pretriggering and
//     start triggering...
//
//  Set triggers
Stream->SetPretriggerType(TriggerManager::ptCounted);
Stream->SetStartTriggerType(TriggerManager::stThreshold);
//
//  Configure Pretrigger to 500 counts
CountedPretriggerPtr(Stream->Pretrigger())->PretriggerCounts(500);
```

```
//
//  Set start threshold to .1 volts
VoltageThresholdPtr(Stream->StartTrigger())->ThresholdLevel(.5);
ConfigurableTriggerPtr(Stream->StartTrigger())->Type(ttEdge);
ConfigurableTriggerPtr(Stream->StartTrigger())->Polarity(tpPositive);
```

Counted Pretriggering allows the preservation of data samples before the start trigger fires. Normally the first data point read was taken just after the start trigger fires. With Counted Pretriggering, N samples before the trigger fires are output when the trigger fires. In the example below the pretrigger is set to return 500 samples from before the trigger.

The AD40 Threshold mode supports two interfaces: VoltageThreshold and ConfigurableTrigger. VoltageThreshold configuration allows the threshold to be set in volts with the ThresholdLevel() method. In the above example the hardware is configured to trigger at half a volt. The ConfigurableTrigger UI interface allows signals to be specified as Type edge or level, and Polarity positive or negative. In the example above, the trigger is configured for a positive-going edge. This means that when the signal crosses the threshold from below to above, the start trigger will fire. A crossing from above the threshold to below it will not fire the trigger.

Each line can be read from the inside out. Stream->StartTrigger() returns the current start trigger UI object. This object is input into the trigger conversion function ConfigurableTriggerPtr() and converted into the ConfigurableTrigger interface. Finally, the Type() method of this class is called to set the trigger type to ttEdge.

The trigger modes a module supports and the UI interfaces its supported modules support are very module dependent. It is quite common to have to use several conversion functions to configure a trigger mode. It is also common for a trigger to be unconfigurable, exposing no trigger UI classes. Similarly, many modules support several triggering modes. Other modules support only the default, unconfigurable combination of no Pretriggering, Always start, Never stop, and no Retriggering. The description of the modes a module supports and the UI interfaces a module supports in each mode are listed in the online help with the description of each module.

Interrupt Handling

In DSP/BIOS, all hardware interrupts are intended to be managed by a DSP/BIOS hardware manager. This manager allows user functions to be called as part of the interrupt process while still cooperating with DSP/BIOS. As a part of the configuration process, the user can direct the HWI manager to call a user function.

Interrupts in a C++ Environment
In a system using C++, this means of attaching interrupts leads to several difficulties. A minor problem is that of name-mangling. C++ creates a new name for every function created in order to allow overloaded functions. The DSP/BIOS configuration does not understand the new name and results in a linker error. There is a simple work-around for this:

```
extern "C"
{
    void  MyHandlerFunction( void * arg );
}
```

This declares to the compiler to create a standard C symbol name for this function (_MyHandlerFunction) which can be used by to the DSP/BIOS configuration tool.

A more fundamental problem is that this mechanism does not allow the interrupt handling function to be changed during the life of the program. Also, this handler function may not be a class member function. This restriction can make designing a class object that handles interrupts awkward.

The Pismo Solution
The solution implemented in the Pismo environment is to take over all interrupt handling by providing a full set of standard handlers. The user then never needs to work in the CDB editor to provide handlers. The standard Pismo handlers contain code that will call a user's installed interrupt handler function if one is provided. While this adds a small amount of latency to the interrupt, the DSP/BIOS overhead per interrupt call is still much greater and dominates the total time per interrupt. In general, the BIOS environment is not suited for extremely high interrupt rates. Luckily, the use of DMA to acquire data from FIFOs on peripherals means that high rate interrupt handlers are not needed.

Pismo uses a special object, a Binder, to group a handler function and its arguments in a way that can be properly called by the standard handler. One form of Binder is used to attach a stand-alone function and its arguments, another form allows the binding of an Object, a member function of that object, and its arguments. This form of binder can allow a class object instance variable to act as a handler for interrupts. Here is an example from the Messages example of defining a binder for a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

And attaching the binder to an interrupt:

```
//  Set up a real time clock to send commands to host on
//    Target channel...
Irq Timer0( intTimer0 );
Timer0.Install( TimerBinder );
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

In the example, TimerBinder is an object that collects the handler function, OnTimerFired, and its argument, 0. This object is passed into an Irq object associated with the TCLK0 interrupt. When the timer interrupt fires, the handler will be called with its argument. The binder is a template, allowing any type of argument to be used with an interrupt handler.

Class Irq

Class Irq is an object that can be created to manage a specific interrupt. It has functions to set, clear, enable and disable the interrupt and also allows a handler to be installed that will be called whenever the interrupt fires. In the above code, see how all functions involving the interrupt were encapsulated in the methods of the Timer0 class object.

Interrupt Lock Classes
A common need in a program is the ability to disable a particular interrupt, or all interrupts, in a portion of the program. The standard means of standalone functions (an disable followed by a enable interrupts) has a few problems. The first is that the means does not nest well. If a function blocking interrupts is nested in a second one, interrupts will be re-enabled at the wrong time. A second is that if the function has multiple return paths, each must have the re-enable code in it. The introduction of C++ exceptions makes this problem even worse.

The Pismo library provides a set of class objects that meet this problem. These lock objects disable a particular interrupt or all interrupts in a region and restore the state to what it was on entry when the

lock object is destroyed. If the object is created on the stack, any means of exiting the block in which the object is defined will cause the cleanup code to be called. Calls to these objects properly nest as well.

Table 8. Interrupt Lock Classes

Lock Class	Interrupts Affected	TI Class Library
InterruptLock	One IRQ	CSL.
GlobalIntLock	All interrupts	CSL.
HwiGlobalIntLock	All interrupts	DSP/BIOS.

Interrupt Binder Templates

The Binder system can be thought of as a more flexible and powerful version of a function pointer variable, allowing a user callback function to be called indirectly without knowing more than the interface to the function. Since the binder objects are templates, the type of the function and its arguments are not fixed but can be of any type. Also, member functions can be bound to an interrupt, which a callback function can never do.

The Binder system is powerful, yet in practice is quite simple to use. This system illustrates the power of the C++ language to contain a complicated system in a simple-to-use package.

Class InterruptHandler
This class is a base class for the ClassMemberHandler and FunctionHandler templates. It provides the interface the Pismo system uses to call the interrupt handler.

Class ClassMemberHandler Template
This template allows the binding of a member function of a class object with the object to call and an argument of any type. In this example the IsrHandler class is bound to a timer interrupt:

```
class IsrHandler
{
public:
    IsrHandler()
        : Binder(*this, &IsrHandler::MyHandler, &Tally), Tally(0)  { }
    ClassMemberHandler<IsrHandler, unsigned int *> Binder;

    void MyHandler(unsigned int * tally)
```

```
    {
        *tally += 1;

        if ((*tally & 0x7f) == 0)
            rtdx << "Isr tally: " << *tally << endl;
    }


private:
    // Data
    unsigned int    Tally;
};

// Instantiate a concrete instance of above class..
IsrHandler  Isr;

void IIMain()
{
    // Dynamically create an Irq object tripped from onchip timer 0
    Irq Timer0( intTimer0 );
    // Bind and install the interrupt vector
    Timer0.Install( Isr.Binder );
    // Program onchip timer 0 to signal at 100 Hz
    Timer0.Enable( false );
    DspClock Clock(100, 150, true, 0);
    Timer0.Enable( true );

    // Use RTDX event log to monitor progress
    rtdx.Enabled(true);

    rtdx << "Message from within IIMain,,,"<< endl;

    // Go to sleep...
    while (1)
        TSK_yield();
}
```

In the above example, the handler uses a int * argument to pass out information from the interrupt routine.

Class FunctionHandler Template

This template allows the binding of stand-alone function with an argument of any type. In this example the OnTimerFired function is bound to a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

This is the installation of the handler in the program:

```
    //  Set up a real time clock to send commands to host on
    //    Target channel...
    Irq Timer0( intTimer0 );
    Timer0.Install( TimerBinder );
```

```
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

EDMA and QDMA Handling

The TI C6000 processor supports a rich, powerful DMA engine to move data without CPU intervention. There are two kinds of DMA allowed. One, EDMA is full featured but can take some time to set up. QDMA is TI's facility for quick DMA movement of data. It is similar to a normal DMA transfer except that it is software triggered and performs only a single transfer. No linking of blocks is permitted with QDMA. It also is faster to initiate as only a few registers need to be set to start a new transfer.

Both kinds of DMA use a set of registers to define the configuration of a DMA transfer. By properly configuring the settings, many different transfer types can be performed, such as interleaved data, two dimensional arrays, and so on. See the TI Peripheral Library guide for more information on configuring EDMA and QDMA.

The QDMA has a single set of configuration registers, so only one QDMA may be in progress at the same time. The EDMA has a pool of blocks that may be used to define simultaneous, complex transfers.

Class DmaSettings
The DmaSettings class manages an image of the settings registers used to configure a QDMA or EDMA transfer. It provides properties to read and set the individual fields of the registers, saving the user the effort of masking bits and shifting data. It even provides functions that preconfigure some commonly used transfers, saving even more programmer effort.

The following code fragment shows how the setter functions are used to set up for a transfer. The DmaSettings class returns a reference to self on all setter functions, allowing multiple parameters to be set on a single line:

```
DmaSettings Cfg;

Cfg.Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit)
Cfg.SourceIncr(DmaSettings::Incr).DestinationIncr(DmaSettings::Incr);
Cfg.TCInt(true).TCCode(1).FrameSync(true);
Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
Cfg.ElementCount(50).ElementIndex(1);
Cfg.FrameCount(0).FrameIndex(1);
```

Class Qdma
This class manages the posting of Qdma requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

As with all DMA objects, the Qdma object uses an internal DmaSettings object to define the transfer. The Settings() method provides access to the object to allow calling the DmaSettings classes own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
    // Q is a Qdma object, here we change the destination address
    Q.Settings().DestinationAddr((int)(dest_array+0x10));
```

For QDMA, a transfer is initiated when the parameters are loaded into the QDMA registers. This is performed by the Submit() method, which starts the preconfigured transaction, or loads the passed in configuration and submits it.

Only one Qdma transfer may be active in the system at one time. Multi-threaded applications must arbitrate Qdmas as appropriate.

If a terminal count interrupt is not used, a call for WaitForComplete() will delay until the completion occurs. TestComplete() will return a flag that can be used to check completion without blocking.

Qdma transfers may be configured to generate Terminal Count interrupts on completion of the transfer. Which TC bit is signaled is configured in the settings block.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the TcIntInstall() method. The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See " Interrupt Binder Templates " on page 59.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second function, TcIntDeinstall() removes any installed handler.

Once installed, TC interrupts may be enabled or disabled by a call to TcIntEnable().

The following example shows a full Qdma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void IIMain()
{

    DmaSettings Cfg;
    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(0);
    Cfg.SourceAddr((int)src_array).DestinationAddr((int)dest_array);
    Cfg.ElementCount(100).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);
    Qdma Q(Cfg);
```

```
        // This QDMA operation will trip a terminal count interrupt when
        // all data has been moved.
        Q.TcIntInstall( Isr.Binder );
        InitArrays();
        Q.TcIntEnable(true);
        qdma_not_done = true;
        Q.Submit();
        while (qdma_not_done)
            ;
    }
```

Class Edma

This class manages the posting of EDMA requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

An additional feature of EDMA is the ability to build complicated transfers by linking EDMA transfer blocks or by chaining EDMA transfers together.

For more information on EDMA, see the TI Peripheral Guide.

As with all DMA objects, the Edma object uses one or more internal DmaSettings object to define the transfer. One block is allocated for the primary transfer, and one for each linked block. The Settings() method provides access to the primary transfer block's settings object. The LinkSettings() similarly allows to one of the link blocks' DmaSettings object. Each of these can be used to call DmaSetting's own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
        // Ed is a Edma object, here we change the destination address
        Ed.Settings().DestinationAddr((int)(dest_array+0x10));
```

The EDMA transfer can be attached to one of a number of channels. To attach an EDMA to a hardware interrupt, use the channel with the same number as the hardware interrupt. For example, to attach an EDMA to external interrupt 4, use the EDMA channel 4.

For EDMA, before a transfer can be initiated, the parameters are loaded into the EDMA PRAM registers. This is performed by the Submit() method, which loads the PRAM with the transfer information. Unlike QDMA, this does not start the transfer itself. The transfer will be initiated when the associated hardware interrupt occurs. If using software triggering, use the Set() function to initiate a transfer. One Set() call is required for each link block in the transfer.

Each Edma transfer allocates blocks from the PRAM pool to configure its Link blocks. These blocks are a limited resource, and the allocation may fail. If the failure occurs, the IsValid() function will return false.

If a terminal count interrupt is not used, a call for WaitForComplete() will delay until the completion occurs. TestComplete() will return a flag that can be used to check completion without blocking.

Edma transfers may be configured to generate Terminal Count interrupts on completion of any and all blocks in the transfer. Which TC bit is signaled is configured in each settings block. This means there can be different handlers for different blocks in the transfer.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the TcIntInstall() or LinkTcIntInstall() method. The Link function is used to install a handler for one of the link blocks as opposed to the primary block.

The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See " Interrupt Binder Templates " on page 59.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second pair of functions, TcIntDeinstall() and LinkTcIntDeinstall() removes any installed handler for the TC bit used by the block.

Once installed, TC interrupts for the entire transfer may be enabled or disabled by a call to TcIntEnable().

The following example shows a full Edma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void EdmaTest()
{
    Edma Ed;

    Ed.Settings().Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit);
    Ed.Settings().ElementIndex(1).ElementCount(50).FrameIndex(1).FrameCount(0);
    Ed.Settings().TCInt(true).TCCode(1).FrameSync(true);
    Ed.Settings().SourceAddr(int(&src_array[0])).SourceIncr(DmaSettings::Incr);
    Ed.Settings().DestinationAddr(dest_array).DestinationIncr(DmaSettings::Incr);
    //
    //  Define a linked
    DmaSettings Cfg;

    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(1).FrameSync(true);
    Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
    Cfg.ElementCount(50).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);

    Ed.AddLink(Cfg);
    Ed.LinkTcIntInstall( 0, Isr.Binder );
    Ed.TcIntClear();

    // This EDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    InitArrays();
    Ed.TcIntEnable(true);
```

```
    qdma_not_done = true;
    Ed.Submit();
    // We software-initiate the EDMA here, but if this EDMA were using EINT4..7,
    // then an external int hardware pulse would remove need for Ed.Set, below
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
    //
    //  Transfer the second transfer block...
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
}
```

The above example sets up a two block linked transfer triggered by software. A TC Interrupt is configured to signal the completion of each block in the transfer. The mainline waits for each block transfer to finish, as notified by the interrupt handler. Then the next block transfer is triggered by a second call to Set(). The Cache functions are required to assure that the cache and memory contents are back in synchronization.

Linked and Chained blocks

EDMA transfers may span multiple transfer blocks. On the completion of the primary transfer, the first link block is loaded into the primary block and initiated. When this block completes, the next linked block is loaded, and so on. A link block can form a loop, but it is important to remember that the primary block can never be part of a loop. Since it is overwritten by the first linked transfer, this transfer can only occur once. Because of this to make a loop of two transfers requires three blocks to be configured. The primary block contains the first transfer, the first link the second transfer, and the third is a repeat of the first transfer that is linked back to the first link block.

Link blocks are allocated by a call to AddLink(). This call automatically configures the preceding block to link to this newly added block. It returns the index of the newly added block that can be used in order to configure the link block. To form a closed loop in a block chain, call LinkBackTo(). This connects the final block in the chain back to the block whose index is given in the argument.

Transfer chaining is a mechanism for having a transfer trigger another on completion. The ChainTo() and ChainEnable() methods set up a chaining relation between two transfers. Note that on the TI C671x processor, the second transfer must be configured on channels 8-11.

Class EdmaMaster

This class acts as a holder for functions and information common to all EDMA interrupts instead of associated with a single EDMA channel. Only one instance of EdmaMaster is created at program initialization. It is accessed by calling the static member function EdmaMaster::Object().

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. The method ClearPram() clears all the PRAM blocks in a single operation.

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma

blocks and not releasing them. Also available are functions to give access to the area at the end of the PRAM that is not used by the system. This scratchpad memory might be of use as a shared memory pool in an application.

P25 Example Programs

Under **P25M\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under Borland C++ Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.x and DSP/BIOS.

Table 9. P25 Example Programs

Example	Host	Target	Illustrates
AEcho	terminal emulator	DSP/BIOS	Use of DSP/BIOS drivers; Analog output driven from analog input
AnalogIn	terminal emulator	DSP/BIOS	Analog capture into DSP memory. Samples sent to host and saved for analysis.
AnalogOut	terminal emulator	DSP/BIOS	Analog waveform playback from pre-calculated buffer in DSP memory.
ASnap	BCB, VC8 (.NET)	DSP/BIOS	Full-rate analog capture to memory, then send to host via busmastering. Illustrates data capture and target/host communication with commands and data packets.
CommonCDB	N/A	N/A	Shared CDB file for all examples.
DataXfr	BCB, VC8 (.NET)	DSP/BIOS	Transfer of Data packets between a target program and the list
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
FftFloat	terminal emulator	DSP/BIOS	Use of floating point FFTs
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
FirFloat	terminal emulator	DSP/BIOS	Use of FIR filters
Servo	terminal emulator	DSP/BIOS	Use of the PISMO Servo class for servo application
Rom	VC8 (.NET)	DSP/BIOS	Read/Write Flash EEPROM contents containing calibration coefficients and board characteristics.
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.

Example	Host	Target	Illustrates
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.
ZbtRam	terminal emulator	DSP/BIOS	Test ZBT ram interface from DSP

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job, or at least refer to the examples to see how some functions are done. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Peripherals

P25M features four channels of 25 MSPS, 16-bit A/D and four channels of 50 MSPS, 16-bit D/A as its analog IO subsystem. The analog directly connects to the application FPGA, allowing complete flexibility in controlling the acquisition and generation of waveforms. Low noise clock generation and error correction logic in the FPGA support the analog IO.

Other peripheral devices on the P25M include two digital IO ports, sample clock generation and control, and triggering controls. Many of these features are implemented in the FPGA logic and may be modified by reprogramming the FPGA. The features described in this manual are those present in the standard FrameWork Logic for the P25M.

Analog Input

The analog inputs are four identical channels of 16-bit, 25MSPS A/D with programmable gain and signal conditioning. All channels are synchronous and acquire data simultaneously. Each channel has an SMB input connector and is 50 ohm terminated. The input is then scaled and filtered before the A/D in the analog signal conditioning circuitry on the card. After the A/D, the digitized signals are captured by the FPGA which performs error corrected and inserts the points into a FIFO.

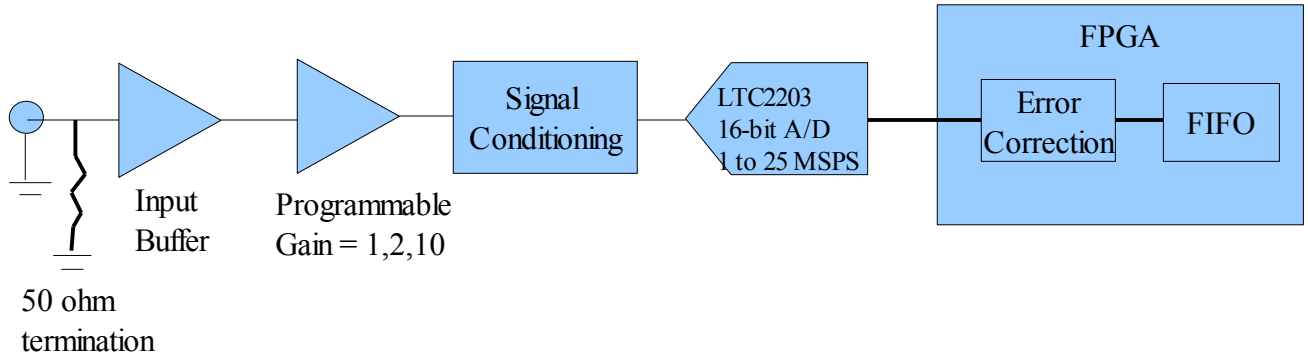


Figure 7. A/D Signal Flow

Analog Input Characteristics

The analog input ranges are shown in the following table for each of the gain ranges provided on the standard card. Custom input ranges can be ordered that will be configured on the hardware according to your specifications.

Table 10. Standard Analog Input Ranges

Input Range	Programmable Gain
-2VDC to +2VDC	1
-1VDC to +1VDC	2
-200mVDC to +200mVDC	10

The analog inputs have a 50 ohm input impedance that matches the cable and connectors to minimize signal reflections and distortions due to impedance mismatches. The signal driving the input MUST be able to drive a 50 ohm input for the full input range of +/-2V. The 50 ohm input termination can be removed so that the input is >1M ohm input, just be aware that the cabling must be tuned to match the input characteristics in the same way an oscilloscope probe is tuned.

!Note : the maximum input voltage should be limited to +/-5VDC to prevent damage.

Programmable Gain

Each P25M has independent programmable gain control that provides gains of 1,2,and 10 in the standard configuration. The gain control circuitry is controlled directly from the application logic and may be set either by the DSP or controlled by the application logic in custom logic designs.

Table 11. Programmable Gain Settings

Input Range	Programmable Gain	Gain Bits 1..0
-2VDC to +2VDC	1	01
-1VDC to +1VDC	2	10
-200mVDC to +200mVDC	10	00
Test Mode – Ground input to filter	-	11

Each gain setting is calibrated during factory test and has unique error correction coefficients that must be loaded when the gain mode is changed. Consult the software documentation for information about controlling the gain and error correction functions from DSP programs.

Analog Input Bandwidth

Analog input bandwidth is generally flat to 10 MHz, with a -3 dB rolloff at 12.5 MHz. The analog signal conditioning electronics limits the bandwidth to remove out-of-band noise that reduces signal quality in most applications. If a modification to the input response is required for your application, contact technical support.

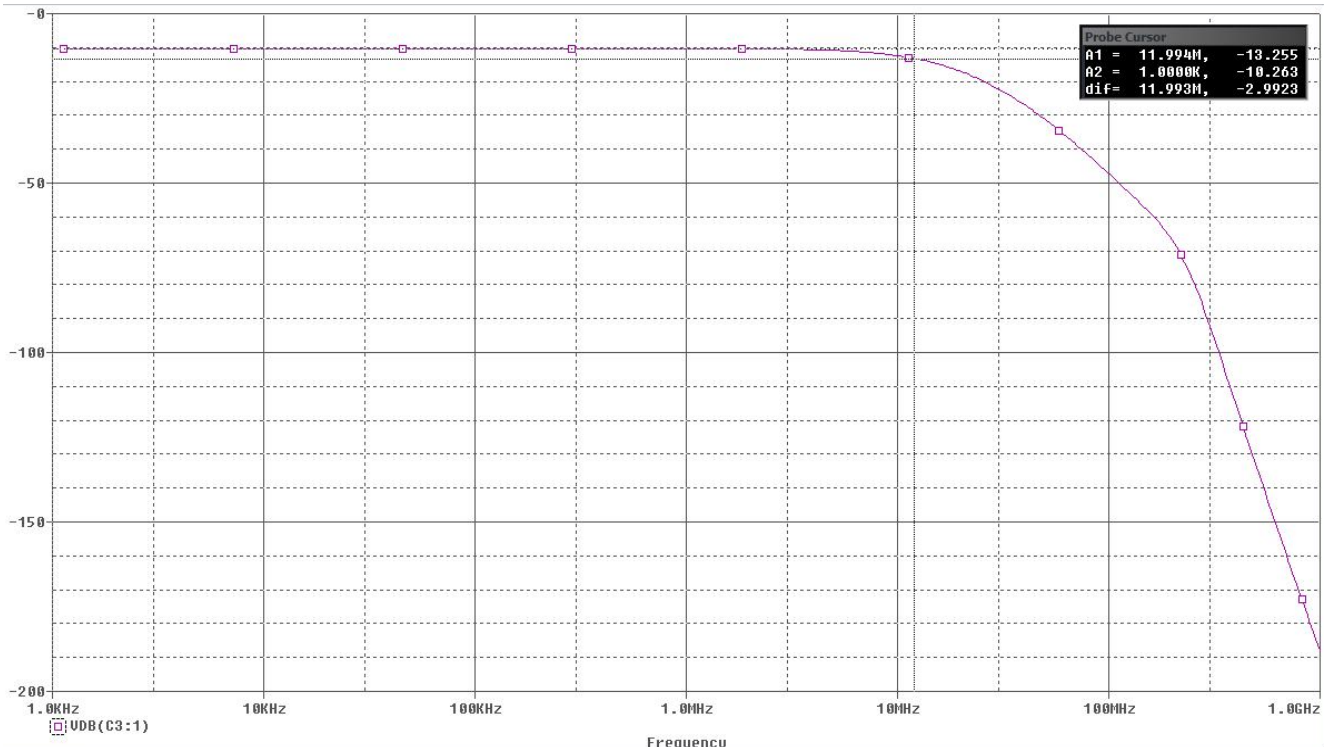


Figure 8. A/D Input Frequency Response

A/D Device

The P25M uses a Linear Technology LTC2203 device for each of the four A/D channels. This is a 25 MSPS, 16-bit device. The A/D is configured on the P25M to provide 2's complement output numbers.

Table 12. A/D output codes

Input Voltage	Output Code	Notes
- Full Scale (-2V for gain = 1)	X"8000"	Most negative number
0V	X"0000"	zero
+ Full Scale (+2V for gain = 1)	X"7FFF"	Most positive number

The LTC2203 A/D has other features that are configured on the P25M as shown here. These are set at the factory on the P25M.

Table 13. A/D Configuration Settings

Feature	Setting	Effect
Reference Voltage	2.5V	Input to A/D is 1.25V +/-1.25V.
Randomization	On	Output bits from the A/D are exclusive-ORed with bit 0 to reduce digital noise. The FPGA logic inverses the bit randomization so that the A/D numbers are as shown in table 3.
Dither	On	The A/D has internal dithering to improve AC noise performance. No effect on output data.
Numeric Format	2's Complement	Output numbers are in 2's complement format.

The LTC2203 has a minimum sample rate of 1 MSPS. This limitation is due to the sample and hold in the device itself. For sample rates below 1 MSPS, the decimation feature in the logic should be used. This allows the A/D to run at a higher rate because the logic decimates the data down to the desired output data rate. It is generally good practice to run the A/D at 10 MSPS or greater because of the improved performance and use the decimation feature since the decimation is done digitally and is therefore noise-free. Maximum sample rate is 25 MSPS. While it may be possible to run the A/D faster, performance has not be verified at higher rates.

Calibration and Error Correction

The P25M includes digital error correction in the FPGA for calibration. The error correction logic provides a first-order, real-time correction to each A/D sample as it is collected to remove offset errors and correct for gain error. The error correction is simply

$y = mx+b$; m = gain correction coefficient, b = offset correction coefficient, x = current sample

and is performed with saturating math. The error correction coefficients are

Table 14. Analog Input Error Correction Coefficients

Coefficient	Size	Notes	Maximum Value Allowed During Factory Test
Gain	18-bit	X"10000" = 1 (default value)	2%
Offset	16-bit	X"0000" = 0 (default value)	+/-300 counts

During factory test, error correction coefficients for each card are determined using a series of measurements against calibration sources (NIST traceable). These error correction coefficients are written to a non-volatile memory device on each card. When the A/D section is initialized, the error correction coefficients must be written to registers in the FPGA. This initialization process is performed by the DSP as part of the A/D driver code.

A common question about the digital error correction is how much analog range is lost because the errors are not corrected before digitizing. At factory test, the maximum acceptable values are limited as shown in the table so that any channel requiring excessive correction is rejected. The analog circuit design is accurate and repeatable enough so that over th 8 years that Innovative has used this method of error correction, we have found that only truly defective channels fail this test. This is partly because we use tight tolerance components so that almost no analog input range is sacrificed. The benefits of long term stability and noise-free error correction vastly outweigh the negligible amount of input range that is lost.

A/D FIFO

In the standard interface logic, the A/D samples are stored in a FIFO after error correction. All A/D channels are merged into a single FIFO that can be read by the DSP. Data samples are stored into the FIFO in the format shown here.

Table 15. A/D Data Ordering in the FIFO

FIFO word	Bits 31..16	Bits 15..0
0	Channel 1(t)	Channel 0(t)
1	Channel 3(t)	Channel 2(t)
2	Channel 1(t+1)	Channel 0(t+1)
3	Channel 3(t+1)	Channel 2(t+1)

where t is the sample number.

The FIFO level is used as an interrupt to the DSP that is used either as a CPU or DMA interrupt. The FIFO interrupt level is programmable by the DSP and is set during initialization. For data acquisition applications, it is more efficient to move larger data sets on each interrupt; servo applications usually move data each sample period for lowest latency.

Channels may be enabled for each A/D 0/1 and 2/3 pair. This does not power down the A/Ds but rather discards the samples in the logic. Disabling an used channel pair reduces the data rate to the DSP by 25 MW/s, which is about 33% of the maximum DSP transfer capacity.

A/D Sample Rate Restrictions

The P25M has 4 channels of 25 MSPS, 16-bit A/D that yields an aggregate data rate 200 MB/s when running all channels at full rate. At this maximum rate, must be processed in the FPGA or stored into the 2MB SRAM attached to the FPGA. The DSP is too slow to process this data because of data transfer rates to the DSP and the processing rate of the DSP itself. Continuous real-time data at 25 MSPS per A/D channel is easily processed in the FPGA or stored to SRAM. FPGA development support tools, including MATLAB tools, are discussed in the *FrameWork Logic User Guide* and *FrameWork Logic MATLAB Guide.*

Continuous data flow to the DSP at 200MB/s is only possible to internal memory when all other bus activity is minimized. Continuous data flow the DSP DRAM is not possible because of rate restrictions imposed by the DSP DMA controller. The controller must perform a read from the FPGA then a write to DRAM, resulting in 2x increase in bus rate. Since the DSP

EMIF operates at 75 MHz, or 300 MB/s burst rates, full rate data exceeds its capability. Also, the DSP processing rate of 300 MHz is generally too slow to perform continuous real-time signal processing on the full speed data when all channels are used. In general, continuous data rates above 5 MHz per channel should use the FPGA as primary signal processing element.

One method to work with the high rate data is to use triggering to control when the samples are collected and to reduce the overall data flow requirements. The FPGA has 1K sample FIFOs built in the can be filled at the full 25 MSPS rate, so in many cases the application can use triggering controls to collect just the interesting data. The triggering modes are discussed in the triggering section of this manual.

Analog Outputs

The analog outputs are four identical channels of 16-bit, 50 MSPS DAC with signal conditioning. All DAC channels are updated simultaneously. The output samples are pulled from an output FIFO and are then error corrected before the DAC devices. Each channel has an SMB output connector with 50 ohm output impedance.

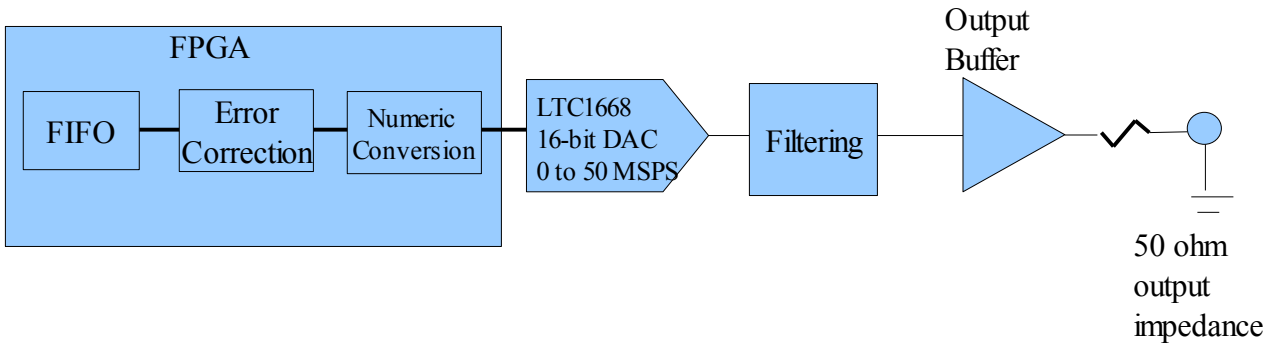


Figure 10. D/A Signal Flow

Analog Output Characteristics

The analog output range is +/- 2VDC on the standard card for a high impedance load. Custom input ranges can be ordered that will be configured on the hardware according to your specifications.

The analog outputs have a 50 ohm output impedance that matches the cable and connectors to minimize signal reflections and distortions due to impedance mismatches. When driving a 50 ohm load the output range is +/- 1VDC.

The 50 ohm output termination can be removed to provide a low impedance output, just be aware that the cabling must be tuned to match the output characteristics in the same way an oscilloscope probe is tuned.

Shorting the outputs to ground may cause damage.

Parameter	Value	Notes
Output Range	+/-2Vp-p	High impedance load
	+/-1Vp-p	50 ohm load
Output Impedance	50 ohms	
Maximum Reverse Voltage	+/-5V	Damage may occur if larger voltages are present

Table 16. DAC Output Characteristics

Analog Output Bandwidth

Analog output bandwidth is limited to -3dB at 12 MHz by an analog filter after the DAC. This filter serves as a reconstruction filter to the DAC and also reduces out-of-band noise on the DAC outputs. The filter is two pairs of complex poles implemented using two sections of an L-C filter.

If a different output filter is required, contact technical support for specifying a custom filter response to match your application requirements.

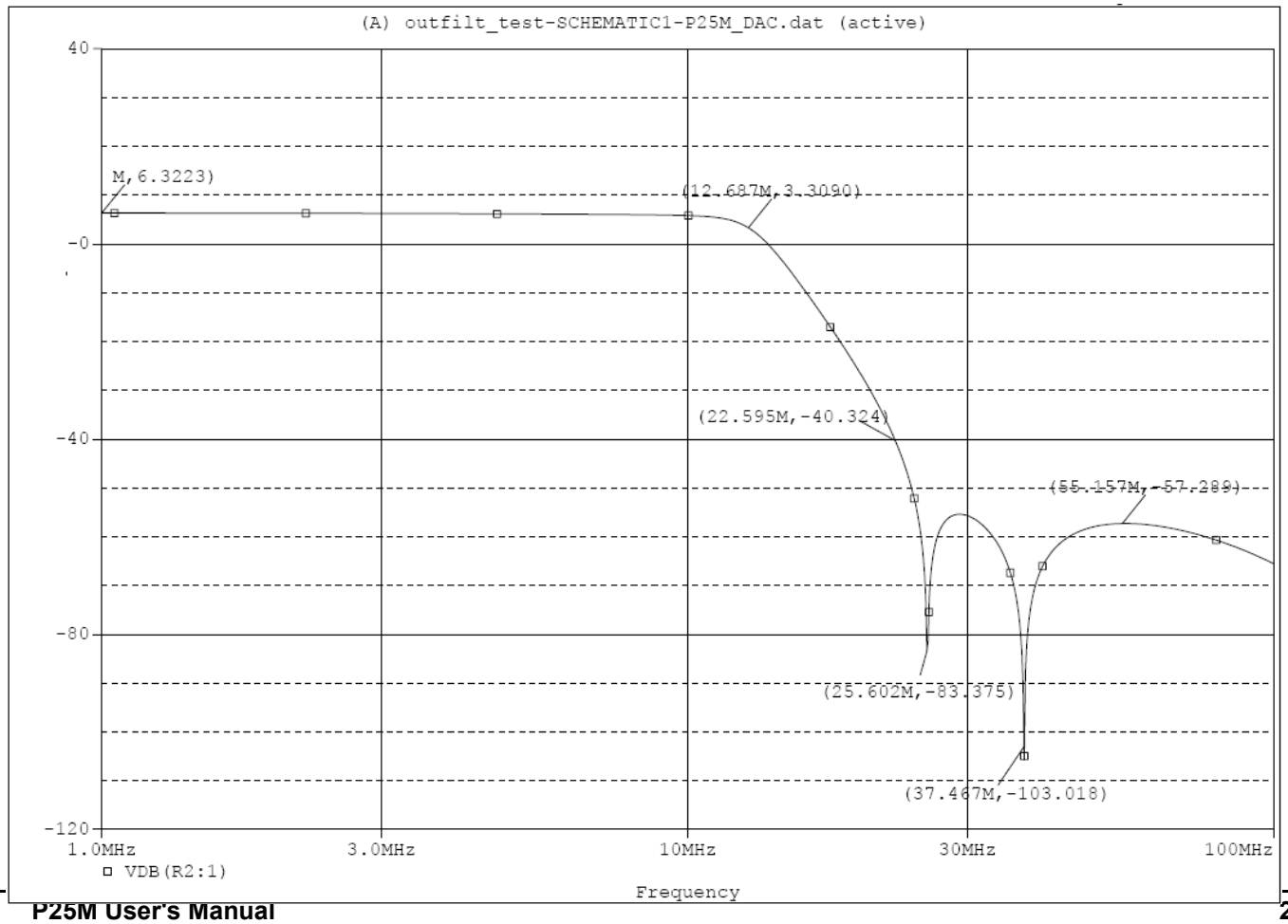


Figure 11. DAC Output Filter Frequency Response

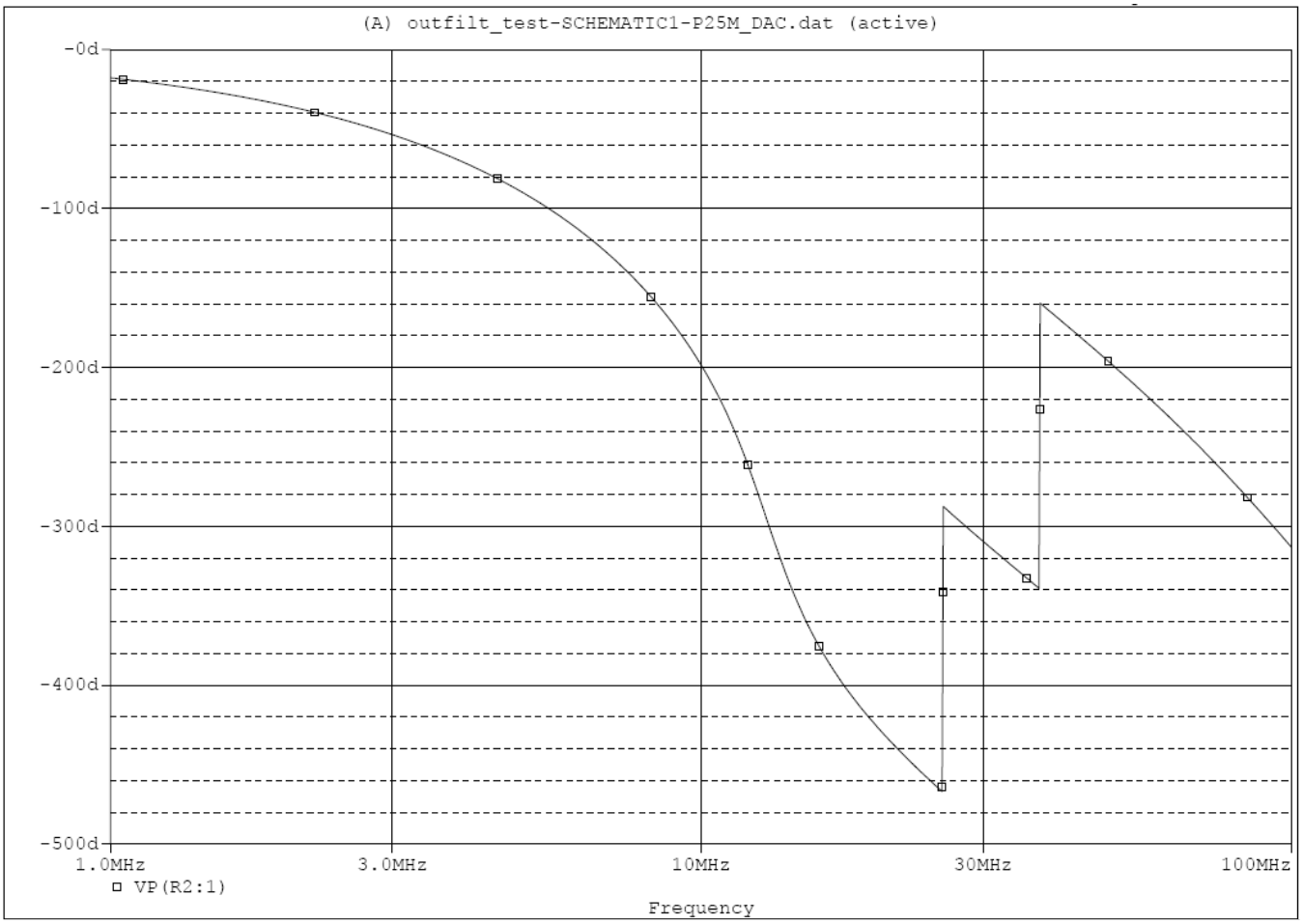


Figure 12. DAC Output Filter Phase Response

DAC Device

The P25M uses a Linear Technology LTC1668 device for each of the four DAC channels. This is a 50 MSPS, 16-bit device. This DAC uses straight binary output coding, but the standard FPGA logic takes 2's complement samples and converts them before output to the DAC.

Table 17. DAC output codes

Input Voltage	Output Code	Notes
- Full Scale (-2V for gain = 1)	X”8000”	Most negative number
0V	X”0000”	zero

Input Voltage	Output Code	Notes
+ Full Scale (+2V for gain = 1)	X"7FFF"	Most positive number

Calibration and Error Correction

The P25M includes digital error correction in the FPGA for calibration. The error correction logic provides a first-order, real-time correction to each DAC sample to compensate for analog offset and gain errors. The error correction is simply

y = mx+b; m = gain correction coefficient, b = offset correction coefficient, x = current sample

and is performed with saturating math. The error correction coefficients are

Table 18. Analog Output Error Correction Coefficients

Coefficient	Size	Notes	Maximum Value Allowed During Factory Test
Gain	18-bit	X"10000" = 1 (default value)	2%
Offset	16-bit	X"0000" = 0 (default value)	+/-300 counts

During factory test, error correction coefficients for each card are determined using a series of measurements against calibration sources (NIST traceable). These error correction coefficients are written to a non-volatile memory device on each card. When the DAC section is initialized, the error correction coefficients must be written to registers in the FPGA.

A common question about the digital error correction is how much analog range is lost because the errors are not corrected before digitizing. At factory test, the maximum acceptable values are limited so that any channel requiring excessive correction is rejected. The analog circuit design is accurate and repeatable enough so that we have found that only truly defective channels fail this test since we choose tight tolerance components so that almost no analog input range is sacrificed. The benefits of long term stability and noise-free error correction vastly outweigh the negligible amount of input range that is lost .

DAC FIFO

In the standard interface logic, the DAC samples are stored in a FIFO. All DAC channels are stored in the same FIFO by the DSP. The FPGA logic pulls samples from the FIFO, corrects the samples to compensate for analog errors, changes the data to straight binary and give the samples to the DACs. Data samples are stored into the FIFO in the format shown here.

Table 19. DAC Data Ordering in the FIFO

FIFO word	Bits 31..16	Bits 15..0
0	Channel 1(t)	Channel 0(t)

FIFO word	Bits 31..16	Bits 15..0
1	Channel 3(t)	Channel 2(t)
2	Channel 1(t+1)	Channel 0(t+1)
3	Channel 3(t+1)	Channel 2(t+1)

The FIFO level is used as an interrupt to the DSP that is used either as a CPU or DMA interrupt. The FIFO interrupt level is programmable by the DSP and is set during initialization. The DAC FIFO interrupt level is usually set such that an interrupt is signaled when the FIFOF has space for data samples. The FIFO is 1K samples in the standard logic. For data acquisition applications, it is more efficient to move larger data sets on each interrupt; servo applications usually move data each sample period for lowest latency.

Channels may be enabled for each DAC 0/1 and 2/3 pair. This does not power down the DACs, it only tells the logic to give samples to the enabled channel pairs. Disabling an used channel pair reduces the data rate from the DSP, saving bus bandwidth.

DAC Sample Rate Restrictions

The P25M has 4 channels of 50 MSPS, 16-bit DAC that yields an aggregate data rate 400 MB/s when running all channels at full rate. At this maximum rate, must be generated in the FPGA or played from the 2MB SRAM attached to the FPGA. The DSP is too slow to process this data because of data transfer rates to the DSP and the processing rate of the DSP itself. Continuous real-time data at 50 MSPS per DAC channel is easily processed in the FPGA or played from SRAM. FPGA development support tools, including MATLAB tools, are discussed in the *FrameWork Logic User Guide* and *FrameWork Logic MATLAB Guide.*

Continuous data flow from the DSP at 400MB/s is NEVER possible because the EMIF operates at 75 MHz, yielding a 300 MB/s maximum data rate. Also, the DSP processing rate of 300 MHz is generally too slow to perform continuous real-time signal processing on the full speed data when all channels are used. In general, continuous data rates above 5 MHz per channel should use the FPGA as primary signal processing element.

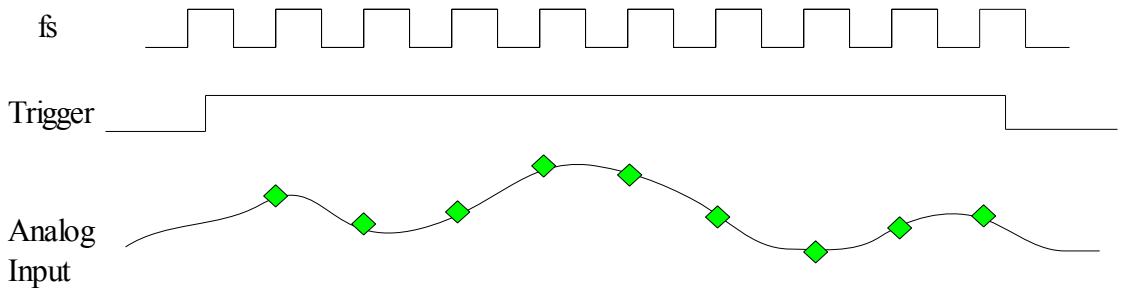
One method to work with the high rate data is to use triggering to control when the samples are played to the DACs to reduce the overall data flow requirements. The FPGA has 1K sample FIFOs built in the can be filled at the full 50 MSPS rate, so in many cases the application can use triggering controls to play out the waveform at the right time, thus reducing the overall data rate required. The triggering modes are discussed in the triggering section of this manual.

Triggering

The P25M has a trigger control component in the FPGA that controls the data acquisition and playback processes. The A/D and DAC trigger controls are independent and symmetric in functionality. Each of the trigger controls allows data to be acquired continuously, or during a specified time, as triggered by either a software or external trigger. Data can also be decimated to reduce data rates.

Trigger Mode	Data Collected/Played Back	Start Trigger	Stop Trigger
Continuous	All enabled channel pairs	Software or rising edge of external trigger	Software or falling edge of external trigger
Framed	N samples blocks of 256 points per block for each of the enabled channel pairs	Software or rising edge of external trigger	Stops when N sample blocks of 256 points per block are collected/played back
Decimation	M points are discarded for every point kept. May be used with either trigger mode.	-	-

Table 1: Trigger Modes



Samples are acquired/played back on rising edges of Fs when trigger is true.

Figure 13. Analog Triggering Timing

As shown in the diagram, samples are captured/played back on rising edges of the sample clock when the trigger is true. The trigger is true in continuous mode after a rising edge on the trigger input, software or external, until a falling edge is found. The trigger is timed against the sample clock and may have a 0 to +1 clock uncertainty for an asynchronous trigger input.

Framed Trigger Mode

In framed mode, the trigger goes false once the programmed number of points N have been collected. Start triggers that occur during a frame trigger are ignored. The maximum number of points per frame is 16,777,216 points. Frame size is always specified on blocks of 256 so that a frame size of two, for example, results in 512 points captured.

Decimation

The data may be decimated by a programmed ratio to reduce the data rate. This mode is usually used when the data rate is less than the minimum sample rate of the A/D (1 MHz). The decimation simply discards M points for every point kept – no

averaging or filtering is used. When decimation is true, the number of points captured in the framed mode is the number of decimated points, in other words the discarded points do not count. Maximum decimation rate is 1/4095.

When decimation is used in the framed trigger mode, the number of points captured is after decimation. The frame count is always the actual number of points inserted into the FIFO.

Digital I/O

The P25M has two digital IO ports, referred to as the DIO and UD port, that provide 64 bits of IO that may be used as inputs or outputs. Both ports are accessed as memory-mapped DSP peripherals and are configured by the DSP. The UD port is implemented in the application logic and may be reconfigured in custom logic applications.

The DIO port is presented on JP3, and the UD port is on connector JP4. See the appendix for the connector pinouts and types.

Software Support

The digital I/O hardware is controlled by the DigitalIo class. Its properties:

Table 20. DigitalIo Class Operations

Function	Type	Description
ClearBit()	Member Ftn.	Sets indicated bit to 0.
Config()	Member Ftn.	Configures bits to input or output
Data()	Property	Broadside Read/Write to DIO.
ReadBit()	Member Ftn.	Reads state of indicated bit
SetBit()	Member Ftn.	Sets indicated bit to 1.
WriteBit()	Member Ftn.	Writes value to indicated bit.

Hardware Implementation

Digital I/O port activity is controlled by the digital I/O configuration control and data registers, one for each port. Port direction is controlled by the configuration control register. The following diagrams gives the register definitions.

Bit	Function
0	DIO bits 7..0 direction control, 1=input, default
1	DIO bits 15..8 direction control, 1=input, default

Bit	Function
2	DIO bits 23..16 direction control, 1=input, default
3	DIO bits 31..24 direction control, 1=input, default
7..4	Not used.
8	Read clock select: 0 = internal (default), 1= external
31..9	not used

Figure 14. DIO Control Register (0x80000000)

Bit Number	31-4	4	3	2	1	0
Bit Field	Reserved	External digital clock	DIR byte 3	DIR byte 2	DIR byte 1	DIR byte 0

Figure 15. UD Digital I/O Configuration Control Register (0x802A0000)

Bit Field Name	Function
DIR byte 0	Direction control for digital I/O bits 7-0 (0 = output, 1 = input, default is input)
DIR byte 1	Direction control for digital I/O bits 15-8 (0 = output, 1 = input, default is input)
DIR byte 2	Direction control for digital I/O bits 23-16 (0 = output, 1 = input, default is input)
DIR byte 3	Direction control for digital I/O bits 31-24 (0 = output, 1 = input, default is input)
Ext Digital Clock	Enable the external digital clock. Default is internal clock.

Figure 16. UD Digital I/O Configuration Control Register Definition (0x802A0000)

Port	Address
DIO	0x80010000
UD DIO	0x80380000

Figure 17. Digital IO Port Addresses

Data may be written to/read from the digital I/O port using the digital I/O port data register. Data written to ports bits which are set for output mode will be latched and driven to the corresponding port pins, while data written to input bits will be ignored. The input DIO may be clocked externally by enabling the external digital clock bit in the appropriate configuration

register. If the internal clock is used, the data is latched at the beginning of any read from the port. Data read from output bits is equal to the last latched bit values (i.e. the last data written to the port by the DSP).

Digital I/O port pins are pulled down to digital ground within the logic device. Consequently, the state of the DIO pins do not change as power is applied to the PC during system start-up. Each DIO bit is ESD and overrange protected.

External signals connected to the digital I/O port bits or timer input pins should be limited to a voltage range between 0 and 3.3V referenced to ground on the digital I/O port connector. Exceeding these limits will cause damage to the P25M hardware.

Digital I/O Timing

The following diagra ock mode (see above for details). Th

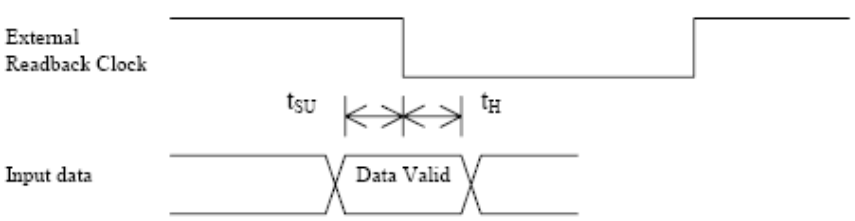


Figure 18. Digita

Table 21. Digital I/O Port Timing Parameters

Parameter	min. (ns)
tSU	5
tH	0

Digital IO Electrical Characteristics

The digital IO pins are TTL compatible, 5V tolerant pins driven by 3.3V logic. The DIO port connects directly to the PCI interface FPGA (XC2S300E). The UD port connects directly to the application FPGA (XC3S1000).

Parameter	Value	Notes
Output Voltage	"1' > 3.0V '0' < 0.8V	For load < +/-12mA
Output Current	+/-12mA	
Input Logic Thresholds	'1' >= 2VDC '0' < 0.8VDC	

Parameter	Value	Notes
Input Impedance	>1M ohm || 15 pF	Excludes cabling

Table 22. Digital IO Electrical Characteristics

Sample Rate Generation and Clocking Controls

For analog sample rate generation, the P25M can use an on-board PLL, external clock, or fixed oscillator. An Analog Devices AD9510 is used for clock generation and distribution that integrates a PLL with clock distribution features. The A/D and D/A devices are clocked directly from the AD9510. The FPGA also receives a copy of the sample clock that is used for data capture and triggering. The FPGA cannot drive the sample clock directly since the sample clock must be very low noise to preserve the A/D and D/A signal quality.

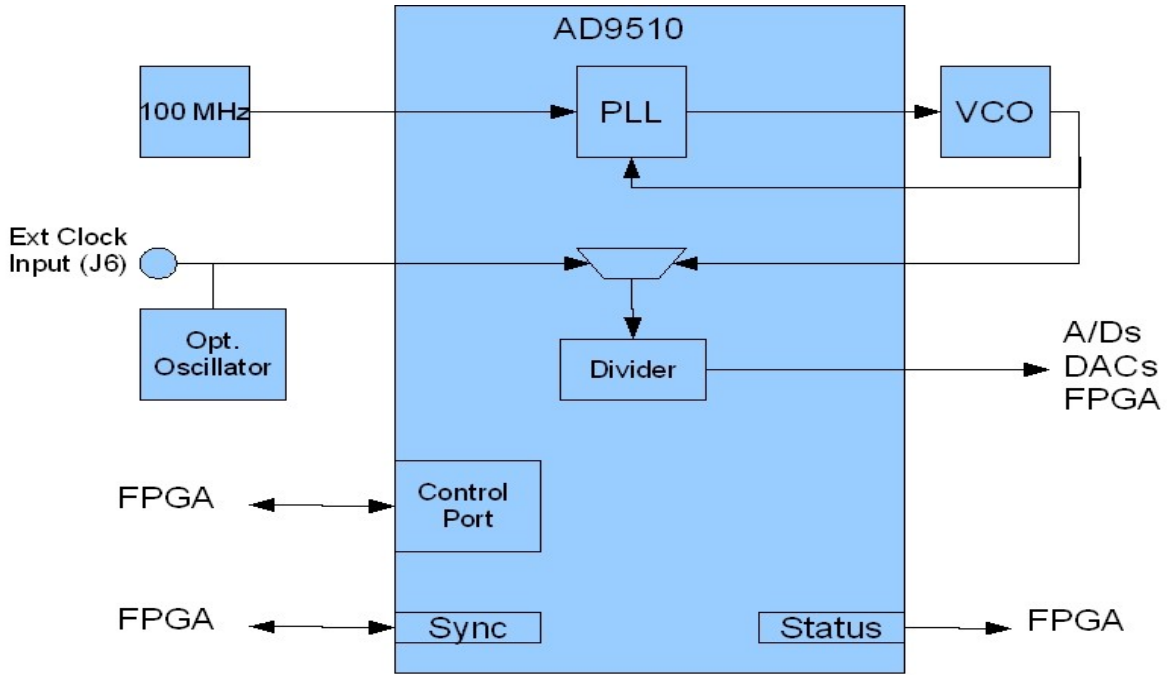


Figure 19. P25M Sample Clock Generation and Distribution Block Diagram

For the P25M, all of the sample clocks are configured as the same frequency and phase. The AD9510 provides 9 output clocks that are used for the A/D, DACs and FPGA. The FPGA clock is configured as LVPECL, the others are all LVCMOS.

External Clock Input

The P25M can use an external clock as the sample rate clock to the analog section. The external clock connector, J6, is a 50 ohm terminated coaxial SMB input. The external clock source should be a low noise source to preserve the analog signal quality on the P25M. As a rule of thumb, the clock should be <1 pS jitter when digitizing a 5 MHz sine input to achieve 90dB SNR, excluding other effects.

The AD9510 must be configured to use CLK1 input when the external clock is used.

Parameter	Min	Typ	Max	Comments
Input Frequency	0		200 MHz	
Input Sensitivity		150 mVp-p (-12 dBm)		
Input Level			2Vp-p (+10 dBm)	Larger inputs may cause damage.
Input Range		-		AC Coupled
Input Resistance	4k	4.8k	5.6k	Ohms
Input Capacitance		15 pF		

Table 23. P25M External Clock Input Requirements

Generating a Sample Clock with the PLL

The PLL is integrated into the AD9510 and is configured to provide clock rates as shown in the following table. This table is based upon a 100 MHz reference clock to the PLL. Custom configurations with a different reference clock can be ordered to meet exact requirements.

The PLL can be tuning range is limited to 100 to 140 MHz when using the VCO standard on the P25M. The PLL has two modes: fixed divider (FD) and dual modulus (DM). The FD mode does not use the A counter.

The tuning equation is

$$F_{vco} = (F_{ref}/R) \times (PB + A)$$

where Fref = 100 MHz

R = 1 to 16383, integers

B= 3 to 8191, integers; 1 = bypass

A= 0 to 63, integers, used only in dual modulus mode

P= reference prescaling, shown in the following table for the two modes

and 100 MHz < Fvco < 140 MHz

Mode (FD = Fixed Divide DM = Dual Modulus)	Value in 0Ah<4:2>	Divide By
FD	000	1
FD	001	2
P = 2 DM	010	P/P + 1 = 2/3
P = 4 DM	011	P/P + 1 = 4/5
P = 8 DM	100	P/P + 1 = 8/9
P = 16 DM	101	P/P + 1 = 16/17
P = 32 DM	110	P/P + 1 = 32/33
FD	111	3

Figure 1. PLL Reference Prescaling

The following table shows a sampling of the PLL output frequencies for a a desired frequency. This table does not show all possible combinations of values and there may be settings closer to a particular frequency of interest. The table is a good starting point. To find the best settings, try adjusting the A value as a fine tune on the other dividers.

$F_{vco} = (F_{ref}/R) \times (PB + A)$

Fs (Mhz)	D	FVCO	Fref (Mhz)	R	PB+A	P	A	B	FS ACTUAL (Mhz)	Error (ppm)
25.000	4	100	100	16383	16383	32	31	511	25.000000	0.0
24.999	4	100	100	16383	16382	32	30	511	24.998474	-21.0
24.950	5	124.75	100	16383	20437	32	21	638	24.949033	-38.8
23.000	6	138	100	16383	22608	32	16	706	22.999451	-23.9
22.000	6	132	100	16383	21625	32	25	675	21.999430	-25.9
21.000	6	126	100	16383	20642	32	2	645	20.999410	-28.1
20.000	6	120	100	16383	19659	32	11	614	19.999390	-30.5
19.000	6	114	100	16383	18676	32	20	583	18.999369	-33.2
18.000	6	108	100	16383	17693	32	29	552	17.999349	-36.2
17.000	6	102	100	16383	16710	32	6	522	16.999329	-39.5
16.000	8	128	100	16383	20970	32	10	655	15.999817	-11.4
15.000	8	120	100	16383	19659	32	11	614	14.999542	-30.5
14.000	8	112	100	16383	18348	32	12	573	13.999268	-52.3
13.000	8	104	100	16383	17038	32	14	532	12.999756	-18.8
12.000	10	120	100	16383	19659	32	11	614	11.999634	-30.5
11.000	10	110	100	16383	18021	32	5	563	10.999817	-16.6
10.000	10	100	100	16383	16383	32	31	511	10.000000	0.0
9.000	12	108	100	16383	17693	32	29	552	8.999674	-36.2
8.000	16	128	100	16383	20970	32	10	655	7.999908	-11.4
7.000	16	112	100	16383	18348	32	12	573	6.999634	-52.3
6.000	22	132	100	16383	21625	32	25	675	5.999845	-25.9
5.000	22	110	100	16383	18021	32	5	563	4.999917	-16.6
4.050	30	121.5	100	16383	19905	32	1	622	4.049930	-17.3
4.001	30	120.03	100	16383	19664	32	16	614	4.000895	-26.2
4.000	30	120	100	16383	19659	32	11	614	3.999878	-30.5

Figure 2. Examples of PLL Output Frequencies

!Note: The minimum sample rate for the A/D devices is 1 MHz. Sample rates below 1 MHz will suffer from degraded performance. For sample rates below 1 MHz, use a higher sample clock rate and decimation in the FPGA.

Fixed Oscillator Option for Sample Clock

The P25M can be optionally configured to use a fixed oscillator. The oscillator, Y1, is installed and C112 is installed (0.01 uF). The oscillator then drives the CLK1 input of the AD9510, requiring that the AD9510 be configured to use Y1 as the source to the clock distribution network in this mode. The output divider in the AD9510 can still be used, allowing the oscillator to be divided by 1 to 32. In this mode, the external clock input is disabled.

The clock oscillator must be a 3.3V, single-ended TTL or CMOS output. The oscillator should have low jitter to preserve the signal quality of the analog inputs, less than 2 ps RMS is typical for a fixed oscillator. One oscillator that fits this requirement is CTS 636L3C025M00000,a 25 MHz oscillator.

Note that C112 is an 0402 size capacitor, located on the back of the board near Y1.

PLL Status

The PLL has a status pin that can be programmed to show when the PLL is locked, or other status information. The status bit function is programmed in the AD9510 (register 8), and is read back at 0x80260000, bit 31.

PLL Interface

The AD9510 is mapped into the DSP memory space for its control port at 0x80260000. Writes to the PLL interface port generate a serial data stream to the PLL that is used to configure the PLL.

Writes to the PLL are performed when the PLL interface port is written to. Reads from the PLL require a two step process consisting of first a write to the PLL register specifying a read at an address, followed by a read from the PLL register that returns the value of the PLL register specified by the address in the PLL word. The PLL is read is a single byte.

This interface is only for configuration, **accesses should be spaced by the DSP to be at least 2 ms apart**.

The PLL interface uses a 24-bit word to communicate with the PLL that specifies a read or write access, the PLL register address and the data byte to transfer. For reads, the data byte is a don't care. The 24-bit word is as follows.

Bits	Function
31..24	Not Used.
23	R/W; 1 = read PLL.
22..15	X“00”
14..8	PLL register address.
7..0	*Data byte (don't care for reads)*

Table 24. PLL Interface Word Format

For reads, the PLL must be written to with a bit 23 as '1' and the address that is to be read, then read from the PLL register. For example, a read to PLL register X”40” would be performed as

Step	Read/Write	Address	Value	Comments
1	Write	X”80260000”	X”00804000”	Set up a read from PLL address X”40”
2	Read	X”80260000”	X”x01303xx”	See format below

Table 25. PLL Read Sequence

The PLL readback word has the following format. The PLL read must be performed before any additional writes are performed.

Bits	Function
31	PLL Status Pin
30..24	“0000000”
23..8	X“1303”
7..0	*Data byte (don't care for reads)*

Table 26. PLL Read Word

Notes About Programming the PLL

The PLL, an Analog Devices AD9510 device, must be initialized prior to use. This device has many configurations that require programming of a large number of registers prior to use. The P25M support software provides PLL configurations that satisfy most applications and should be used if possible.

For custom configurations, the AD9510 data sheet should be consulted. The P25M uses the AD9510 for nine output clocks – one for each A/D and DAC channel plus one for the FPGA. These clocks are connected as shown in the following table. The FrameWork Logic is predicated on the all of the channels operating synchronously, therefore the AD9510 is programmed so that all channels have the same clock output. The A/D and DAC clocks use outputs 4 through 7, with the A/D on the positive output and the DAC on the negative output. The PLL should be programmed to use these outputs with the signal type noted.

Channel	AD9510 Output	Signal Type
FPGA	Channel 0	LVPECL
A/D 0	4 +	CMOS
A/D 1	5+	CMOS
A/D 2	6+	CMOS
A/D 3	7+	CMOS
DAC0	4 -	CMOS
DAC1	5 -	CMOS
DAC2	6 -	CMOS
DAC3	7 -	CMOS

Table 1. PLL Output Assignments

The VCO used with the AD9510 has a tuning range of 100 to 140 MHz and is connected to the CLK2 input to the PLL. The standard reference clock is 100 MHz to the PLL. The output of the PLL section of the AD9510 can therefore be programmed to many numbers in the range of 100 to 140 MHz, that may be subsequently divided in the PLL outputs. The dividers in the clock distribution section of the AD9510can be used to further divide the clock by 1 to 32, with the restriction that some numbers are not used and that the clock is NOT 50% duty cycle for odd numbers.

The external clock and optional fixed oscillator are connected to the CLK1 input. The PLL must be programmed to use one of these two clock sources for the outputs. The clock dividers on the outputs should be programmed to the same divisor to work with the standard logic.

The AD9510 is programmed during initialization of the card. All configuration registers are written, then an update command is sent to the PLL that makes the outputs update simultaneously. After an update, the clock is stable when the PLL status bit indicates a lock, but the A/Ds require additional time to stabilize so a 1 ms period should be allowed for stabilizing the clock.

Serial EEPROM Interface

P25M has a serial EEPROM for storing data such as board identification, calibration coefficients, and other data that needs to be stored permanently on the card. This memory is 16K bits in size. Functions for using the Serial EEPROM are included in the Pismo Toolset that allow the software application programmer to easily write and read from the memory without controlling the low-level interface.

As delivered from the factory, this EEPROM contains the calibration coefficients used for the A/D and D/A error correction.

The serial EEPROM device is an Atmel AT24C16 or equivalent.

! Caution : the serial EEPROM contains the calibration coefficients for the analog and is preprogrammed at factory test. Do not erase these coefficients or calibration will be lost.

Servo Applications

Applications for DSP baseboards can be broadly divided into two categories: Data Acquisition and Servo Control applications. These two types of application require very different methods of behavior by the system that cannot easily be reconciled into a single 'one size fits all' system. Yet the Pismo library needs to support both styles of program in a simple and natural way.

Data Acquisition Applications
Acquisition applications need to acquire large quantities of data, but do not need to process the data immediately. In order to support higher rates, these applications use large buffers, and large chains of these buffers to allow data to be moved from the hardware into memory without CPU intervention. Thus data can be acquired at high rates, but the system only needs to be involved at relatively rare intervals to process a block of data.

These applications are very natural for DMA, and often DMA is used to move this data into memory. In addition, these buffers and the FIFOs in the hardware allow some 'slack' in the system so that the application can fall behind for a short time if it is busy performing some other service. After the completion of this task the system can process the buffers in the queues and in the hardware FIFOs and catch up. As long as enough slack is built into the system, no data loss will result.

Servo Control Applications
In a Servo application, the requirements are polar opposites from the Data Acquisition application. In this case, each event needs to be processed at once, with no delay. This data is analyzed to produce an output update event that is output to the DACs in the system at once. It is therefore vital that data is read into the system and processed without any buffering. Hardware FIFOs are only useful at the single event level -- if you fall behind by even a single event your servo is failing.

Since the amount of data moved from the hardware is so small, DMA is much less useful in this case than it is for the Data Acquisition case. In fact, due to very large latencies in initiating a DMA operation, only quick DMA (QDMA) is a reasonable candidate for use within a servo application. However, QDMA is not utilized within the Pismo servo driver for the following reasons:

1. QDMA requires software initialization in order to initiate data movement. This initialization must be performed each servo interrupt cycle. The amount of time saved by having the DMA engine (more efficiently) move between the peripheral and the servo application is roughly comparable to the amount of time lost performing QDMA setup and cache manipulation functions.

2. QDMA is a limited system resource. The Pismo Buffer classes use QDMA to perform high speed copies. So, use of QDMA within the servo driver would preclude its use elsewhere.

Servoing and DSP/BIOS
DSP/BIOS uses device drivers to provide a uniform, simple interface to analog hardware for input and output. The driver model used is well suited for Data Acquisition applications, since it provides simple means for allocating and passing buffers of data in and out of the device. This driver model is used for all of the streamable hardware provided on the baseboard. Yet because of the strength of the model for the streaming applications, these drivers fail to be able to properly servo in any efficient manner.

The Servo Base Class
The Pismo Library supports servoing by defining a class to perform the basics of a servo operation. The basic servo operation is to acquire data from the analog input, perform some kind of algorithm using the new data to produce an output data event, which is then output to the DACs. The basic servo class handles the details of attaching the interrupt, setting up the hardware, reading the data from the hardware and delivering it to the user's function for processing. After this, it writes the modified data to the DACs.

Using the Servo Class

The ServoBase base class provides a number of methods to simplify the operation of a servoing application. It organizes the configuration of the hardware and the timebases to provide the exact setup needed. It provides a 'callback' hook to allow user code to be called when running the servo in order to provide custom servo processing functionality.

The following table gives the methods that can be overridden, and when they are called:

Function	When Called	Function
Execute()	On each interrupt	Servo calculations

Table 27. The Servo Class Virtual Function

Servo Interrupt Modes

The ServoBase class sets up a configuration where the analog will drive an interrupt on every incoming point, and expect a return event to be loaded very soon thereafter. A key factor in the performance of the servo is the performance of the interrupt and its internal functions.

The ServoBase class provides all the functionality of a 'null' servo...that is, one that performs no modification of the data from input to output, including installation of its own interrupt handler. This handler will read data from the input peripherals into a buffer, call the Execute function in which the user performs custom servo calculations, then transfers the data out to the output peripherals.

ExecuteUser code within the overridden Execute method always executes within HWI interrupt context. This code may call BIOS routines only if the the ServoBase object used within the application is constructed with the UseDispatcher parameter set true. However, setting this flag results in a less- efficient interrupt handler, exhibiting high interrupt latencies than if UseDispatcher is false.

Once in the interrupt, the application software merely manipulates samples within the event buffer whose address is passed as a parameter to the Execute method. Input samples are consumed from this event buffer and output response values are stored into this same buffer. Upon return from the overridden Execute method, the contents of this buffer is automatically written to the output devices.

The use of these methods allows the servo system overhead to be reduced to the minimum possible for this architecture. The Servo example program, included in the Developers Package, demonstrates this technique.

Execute()

The Execute method is overridden with the user's Servo code. The function is very simple -- the data taken from all enabled input peripherals is passed in as an argument, and needs to be overwritten with the data to be written to all enabled output peripherals. The data is passed in the same format as the peripherals produce -- for example, paired data in channel order on the Servo16 module. Similarly, the output format is that which the output peripherals will require. If the channels output are different than those input, or more or fewer, the data is stored left-justified within the event buffer for proper results to be obtained.

This may seem complicated, but it is really rather simple in practice. The job of this function is to take the input data and 'convert' it into an output event using your own algorithm. All other details are taken care of by the system.

Timebases

The Servo object performs analog input and analog output, just as device drivers do, so it needs to allow timebase information to be passed into it. A clock user interface object is manipulated in a fashion identical to the streaming analog drivers to initialize the servo conversion clock.

DAC Delay

The P25M has special timebase support for the servo control applications that delays the conversion of the DACs a fixed amount relative to the conversion clock. The purpose of this delay is to give time for the Servo interrupt processing code to acquire, analyze, and produce the output data for the DAC to output. This value needs to be tuned to match the time it takes your specific application to do this operation, depending on the servo code. The delay is set by the Delay() property of the Servo object.

Channel Selection

The ServoBase objects InputChannels() and OutputChannels() methods are used to obtain access to the channel information objects for the module's input and output peripherals. Generally, the use of these methods is to specify the number of active input and output channels.

Servo Timing

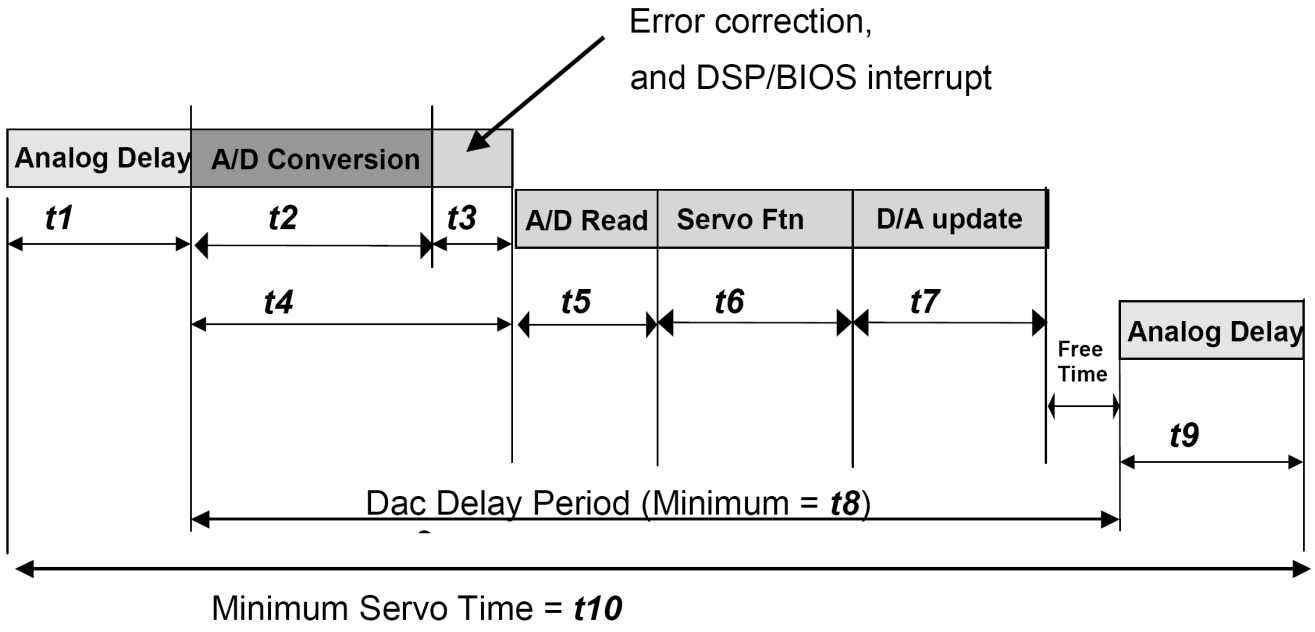


Table 28. Servo Time Measurements

Interval	Purpose	P25M Period in μS (w/Dispatcher)	Period in μS (w/o Dispatcher)
t1	Adc Analog Delay	0.1	
t2	Adc Conversion	0.04	
t3	Logic + Interrupt Time	0.8 + 2.9	0.8 + 1.0
t4	*t2* + *t3*	3.74	1.84
t5	A/D Read Time	0.8 for 1 pair 1.6 for 2 pair	
t6	Servo Calculation Time	?	
t7	D/A Write and Update plus module interrupt acknowledge	2.3 for 1..2 pairs	
t8	Minimum Dac Delay Time	6.84..7.64	4.94..5.74
t9	Dac Analog Delay	.1	
t10	Total Servo Time	7.04..7.84	5.14..5.94

The analog delay inserts a fixed skew between input and output and does not affect the update rate directly.

A Servo Tutorial

The following section walks through an example servo. This example can be found in the distribution in the Servo directory under the Examples directory.

This servo modifies one channel in a simple but visible way to show the servo effect. In actual practice all channels would probably be modified.

The first interesting operation is the derivation of a new ServoBase class.

This example does not utilize the non-dispatcher, to maximize performance. The TestServo class informs the base class that the application will not make BIOS calls within the interrupt by the second 'false' argument to the constructor.

```
//============================================================================
//  class TestServo -- this examples servo
//============================================================================

class TestServo : public ServoBase
{
public:
        TestServo(Omnibus::ModuleSite site)
                : Servo(site, false)
                {}

};

};
```

To derive, you need to create a constructor for the new class. Since the ServoBase base class requires an argument, C++ can not create one for you. The base class needs to be informed of which Omnibus module site is hosting the installed Servo16 I/O module. In this case, we require our servo to have it passed in when it is created. An alternative would be to 'hard code' a value and not require (or allow) the user to change it.

Next, we define our interrupt handler function.

```
        // Overrides
        #pragma CODE_SECTION(".hwi");
        virtual void Execute(volatile short * event, int inputs, int outputs)
        {
                // clip above this amplitude
                const int Threshold = 0x4000;

                if (event[0] > Threshold)
                        event[0] = Threshold;
        }
```

Here we actually define the servo function. The handler is defined as an override for the virtual Execute method. The #pragma places this function onchip for extra efficiency.

The base class passes several important parameters to the application. event is a pointer to a buffer containing an array of 16-bit input samples read from the input peripheral obtained during the most recent conversion. inputs and outputs are the count of active input and output channels, respectively. The body of the method illustrates processing on input channel zero only. The value of input channel zero is clipped to 0x4000 counts. For the Servo16, this clips the input voltage to half-scale. The

clipped value is written back to the event buffer, overwriting the original value. The effect of this handler is to loop-through all inputs unchanged to all outputs, except for channel zero, which is clipped.

The Pismo Library creates a new main() function, IIMain() which is the primary application thread. This thread is responsible for creating, configuring, and running the servo.

```
//--------------------------------------------------------------------------
//  IIMain()  --  Application mainline
//--------------------------------------------------------------------------

void IIMain()
{
        int DacDelay;
        int SampleRate;

        //
        //  Terminal I/O
    cio << init;
    cio.At(Point(25, 0));
    cio << bold << "\7Servo Application\n\n" << normal << endl;

        TestServo ServoIo(Omnibus::mSite0);
```

First, the application instantiates a TestServo object called ServoIo. This provides access to the servo driver, customized with a unique Execute behavior. This driver is Opened, then the internal event buffer is relocated to onchip ram, for enhanced performance.

```
    LoadModule(Omnibus::mSite0, PickModuleType());

        bool status = ServoIo.Open();
        cio << "Servo open " << (status ? "ok" : "failed") << endl;
        if (!status)
                DynamicHang("Servo interface not supported");

        // Locate event buffer onchip
        ServoIo.SegId(1);
```

First, the application obtains a pointer to the timebase used for the analog input. This is a ‘plain old timebase’ which runs continuously with no special triggering. Its rate is set by the property call to Clock->Rate().

```
    cio << "Enter sample rate (<= 100 kHz): " << flush;
    cio >> SampleRate;
    cio << endl;
        SampleRate *= 1000;

        ClockRateUI * Clock = ClockRateUIPtr(ServoIo.Clock());
        if (Clock)
            Clock->Rate(SampleRate);
```

The example allows a variable number of channels be processed. This is accomplished by manipulating the input and output channel control objects accessible through the InputChannels and OutputChannels methods.

```
int MaxChannels = std::min(ServoIo.InputChannels().Channels(),
                                                    ServoIo.OutputChannels().Channels());
int Channels;
cio << "Enter # channels: " << flush;
cio >> Channels;
cio << endl;
Channels = std::min(MaxChannels, Channels);

// Enable all analog input and output channels
ServoIo.InputChannels().EnableChannels(Channels);
ServoIo.OutputChannels().EnableChannels(Channels);
```

The output delay is also adjustable, through use of the Delay method inherited from the base ServoBase class.

```
int MaxDelay = (((1.0/SampleRate) * 1.e9) / 100) * 100;
cio << "Enter the Dac Delay Time (< " << MaxDelay << " ns): " << flush;
cio >> DacDelay;
cio << endl;
//
//  ...Configure delay of DAC clock
ServoIo.Delay(DacDelay);
```

Interrupt latency is increasingly erratic in the presence of other interrupt sources. The example allows the USB and serial port interrupt to be disabled, to improve determinicity.

```
char NoComm;
cio << "Disable communications? " << flush;
cio >> NoComm;
cio << endl;

cio << "\nServo Running... press any key to stop." << endl;
Sleep(300);

// If communications are disabled, max servo interrupt latency is
// reduced, since fewer interrupt conflicts occur
if (std::tolower(NoComm) == 'y')
       DisableCommunications();
```

The driver is open and viable. To start data flow, simply call the Start method. Interrupt processing within the ServoIo Execute method will commence immediately thereafter.

```
// Start processing
ServoIo.Start();
while (!cio.KbdHit())
       {
       Sleep(300);
       cio << "Serviced interrupt: " << ServoIo.Tally() << cr;
       }
```

Data flow may be suspended and resumed through repeated calls to Start and Stop.

```
        // Stop processing
        ServoIo.Stop();
        // Close driver
        status = ServoIo.Close();

    cio << "\nProgram terminating" << endl;
    cio.monitor();
```

The loop continues to run until the user hits a key on the operating panel. The main thread must be kept from terminating to keep the Servo alive. The above code is a simple way to do it.

Then when the loop is finished, we close down and exit:

```
        // Terminate streaming
        status = ServoIo.Close();
```

Debugging Note: It is important that the Close() method be called during debugging, or that the CPU be reset for proper servo results. We have found that if the servo is interrupted and the program reloaded in CodeComposer directly, the servo may be delayed an additional clock cycle. Either performing a full download from the host program or allowing the servo to close avoids this problem.

Conclusion

The power of the Pismo library and the servo application is that the details of configuring the analog are hidden in the objects provided, yet can be configured if necessary. The code that is written is much clearer since the support code is concealed internally. The application is simpler and easier to maintain because the code is almost entirely the important servo algorithm.

Communication to the Host

Overview

Many applications involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they run independently from the host after that.

Other applications need to interact with a host program during the lifetime of the program. This may vary from a small amount of information to acquiring large amounts of data. Some examples:

- Passing parameters to the program at start time
- Receiving progress information and results from the application.
- Passing updated parameters during the run of the program, such as the frequency and amplitude of a wave to be produced on the target.
- Receiving alert information from the target.
- Receiving snapshots of data from the target.
- Sending a sample waveform to be generated to the target.
- Receiving full rate data.
- Sending data to be streamed at full rate.

These different requirements require different levels of support to efficiently accomplish. The simplest method supported is performing file I/O from within Code Composer using either the standard C file functions (which communicate directly through CCS to the Host file system), or via the Innovative terminal emulator, which supports simple data input and control and the sending of text strings to the user in addition to file I/O.

However, the highest level of support is given by the Bus Mastering Interface. This allows sophisticated high-rate transfer of commands and information between the host and target. It requires more software support on the host than the standard I/O does.

CPU Busmastering Interface

Firmware within the logic onboard the P25M is capable of high-speed PCI busmastering to move data between target and host memory. This busmaster facility can be used to transfer data between host and target applications. Enhanced DMA is

used to move data between DSP memory and a logic-based FIFO. The firmware within the baseboard logic moves data between this FIFO and Host PC memory.

CPU Busmastering Implementation

Packet Based Transfers.
The CPU busmaster interface implemented within the the P25M transfers discrete blocks between the source and destination. Each data buffer is transferred completely to the destination in a single operation. Only if several transfers are requested at once will any delay in beginning transmission occur, as multiple requests have to be serialized through a single hardware system.

The data buffers transferred can be of different sizes. Each requested buffer is interrogated for its size and fully transmitted. At the destination, the destination buffer is dynamically re-sized to allow the incoming data to fit. If the buffer given is too small for the data, it will be reallocated to allow the transfer. Reallocating buffers can take some time, for best performance buffers should be pre-sized to be large enough for the largest transfer expected. This will make allocation of buffers at critical times unnecessary.

Blocking Interface
CPU busmastering uses a simple blocking interface for its send and receiving functions. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data has arrived from the data source and transferred into the data buffer before returning. At this point the buffer is ready for use. This blocking allows sequences of transfers managed by a simple sequence of calls to transfer functions.

Since the transfer functions are blocking, they are best avoided in the main user interface thread of a Windows application. The GUI will be appear to be 'frozen' until the transfer has completed. For best results, the data transfer functions should be placed in separate threads on the target and host applications. In fact, each direction of transfer should have its own thread, so that the two directions of transfer can interleave as much as possible. The example programs CpuBmIn and CpuBmOut illustrate the use of separate threads for data transfer.

Maximum Transfer Size
The largest transfer allowed is half of the total size of the Dma Buffer allocated by the INF file when the driver is installed. Half of the memory is dedicated to each direction. The default buffer size in the INF is 0x200000 bytes, so the maximum transfer is 1 Megabyte.

Host (Armada Library) Support for CPU Busmastering

The Host P25M object contains the following two methods to support CPU Busmastering:

```
//
//  Block Transfer System Methods
bool       Send(int Channel, const IntBuffer & Block);
bool       Recv(int Channel, IntBuffer & Block);
```

P25M::Send() sends the contents of a IntBuffer object to the target. All of the data in the IntBuffer is transferred. There is no means of sending a partial buffer. The function will not return until the block has been transferred to the target.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

P25M::Recv() waits for data to arrive from the target, then returns the data in the buffer provided. The IntBuffer will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block (which can degrade real-time performance).

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

Target (Pismo Library) Support for CPU Busmastering

In the Pismo library, the Utility library contains a file PciTransfer.h that contains this class:

```
class PciTransfer : public PciTransferBase
{
public:
               PciTransfer();

               bool  Send(int channel, const Buffer & buffer);
               bool  Recv(int channel, Buffer & Buffer);
};
```

PciTransfer::Send() sends the contents of a Buffer-derived object to the Host. All of the data in the buffer is transferred. There is no means of sending a partial buffer. The function will not return until the block has been transferred to the host.

The use of the base buffer class allows any of the IntBuffer, CharBuffer, FloatBuffer and similar classes to be sent across the interface.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

PciTransfer::Recv() waits for data to arrive from the target, then returns the data in the buffer provided. The Buffer will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

See the ASnap example for an illustration of bus-master usage.

C++ Terminal I/O

The terminal emulator applet is a Host PC application which provides a C++ language-compatible, terminal emulation facility for interacting with the TermIo Pismo library running on an Innovative Integration DSP processor.

Using the terminal emulator, it is possible to develop and debug target DSP code while deferring development in Host application code. By using simple, streaming I/O functions within a target application during development, DSP algorithms can be developed independently from Host applications. Later, when a custom Host application code is written, the DSP standard I/O functions may be deleted from the target application and the target application will no longer be dependent on the emulator or the target TermIo libraries.

Streaming methods such as << and >> are dispatched by the TermIo object to route text and data between the DSP target and the Host terminal terminal emulator applet. Text strings are presented to the user via a terminal emulation window and host key-board input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.

Target Software

All of the features of the terminal are accessed through the two classes TermIo and TermFile. TermIo provides the basic streaming interface which allows text messages to be formatted and streamed out to the terminal as well streaming in strings and numeric values from the terminal for consumption by target application code. The TermFile class provides a mechanism allowing target applications to open host disk files, perform read and write accesses, and subsequently close these files. See the Files.cpp example for illustrative usage of each of these classes and their functions.

Tutorial

Using the terminal during target software development is simple. The global cio object which is automatically instantiated within the Pismo libraries. Use the methods within the cio class to format text strings and then stream them to the UniTerminal applet

```
cio << bold << "7Demonstrate file I/Onn" << normal << endl;
```

Note the use of manipulators, such as bold and normal, to force formatting of the text string as it is streamed to the host. TermIo features many such manipulators to perform functions such as setting text color (setcolor), clearing to end-of-line (clreol), clearing the screen (cls) and so forth. Other manipulators are available to format numeric values as they are streamed to the host. For example, the phrase

```
cio << "Hello" << hex << showbase << 4660 << dec;
```

displays the string "Hello 0x1234" on the console display, converting the integer value 4660 as a hexadecimal number on the target, prior to streaming it to the host. Other manipulators are available providing extensive control over the display of floating point numbers as well as integer values.

It is also frequently necessary to obtain input from an operator during the run-time execution of a target application. For example, it may be necessary to prompt for a sample rate at which analog I/O is to be streamed. The code fragment below illustrates the necessary technique

```
// Prompt the user
cio << " Enter a float: " << flush;
float x2;
// Eat user input
cio >> x2;
```

The stream manipulator >> is overloaded to allow streaming directly into floating point, integer and string variables directly from UniTerminal.

To perform file input and output from within target applications, first instantiate a TermFile object, as below

```
TermFile File;
```

Then, use the TermFile Open method to open the file for access on the host using the desired open attributes

```
if (!File.Open("wave.bin", "w+b"))
    {
    cio << "nOutput file open error - Program terminating!" << endl;
    cio.monitor();
    }
```

This method returns a Boolean indicating success if the file open is successful. To store data into the file or retrieve data from the file, use the Write or Read methods, respectively. For example

```
transferred = File.Write((char*)&Buffer[0], 10000);
```

writes 10000 bytes of Buffer into the disk file. When disk operations have been completed, the file should be closed using the TermFile::Close method.

Developing Host Code

This section describes the Innovative Integration Windows host software development environment for the P25M. Support for controlling the board and the network libraries is provided by the native C++ class libraries, named Malibu, that can be linked into any application. Sample applications show how to use the Malibu Library to initialize the hardware and communicate with to the hardware in operation.

Malibu for the P25M is a native C++ class library. It provides software to support threading, buffer manipulation, and other system functions. The use of native C++ code makes the use of the libraries on any platform straightforward, since the interface for all classes will be identical wherever it is used.

Host software development is directly supported under both the Borland C++ Builder 6.0 and Microsoft MSVC v7.0 environments for generating 32-bit Windows applications. While the supplied examples use the Borland VCL or MSVC foundation classes for the user interface, the Malibu class library itself is not dependent on any particular compiler-specific frameworks.

Please Note: Only Windows application development is currently supported by the Developer's Package. Foreign operating systems are not currently supported.

The Malibu Libraries

Malibu is separated into several individual libraries, organized by the function of the files it. All Malibu libraries and files have an _Mb suffix to their base names, to avoid naming conflicts with other libraries.

Library	Description
Utility_Mb	Software Utility classes and functions.
Hardware_Mb	Hardware control classes for baseboard control.
Kernel_Mb	Driver Kernel Functions (MS VC++ only).
Ethernet_Mb	Ethernet Baseboards and support classes.
Pci_Mb	PCI Bus Baseboards and support classes.
Analysis_Mb	Data analysis and Signal Generation classes.

Interacting with a baseboard is simple. Each baseboard type has a class defined that is used to access the board and control its hardware. When object of this type is created, the library performs the hardware operations needed. When linked to the Malibu libraries, the application automatically controls the physical board.

Malibu supports the exchange of data between the board and the application using the most effective means the hardware supports. On some boards this is PCI busmastering, on others, Ethernet. In either case the library has full support for data exchange. Usually, the I/O is abstracted into a Stream class, which is linked to a baseboard for use. This allows the possibility of multiple modes of communication on a single baseboard.

Data exchange uses objects to hold the data exchanged. Malibu supplies fully featured classes for buffers: IntegerBuffer, FloatBuffer, ShortBuffer and so on. Shorter command packets on some baseboard use a Message class for transporting the information.

Table 29. Some Classes In Malibu

Class Name	Description
P25M	Baseboard support class.
TransferPacketHeader	Command Packet parsing.
MemoryBuffer	Buffer management.
Event	Thread Synchronization.
Thread	Thread class.

Note that this is just a partial listing of available classes within the Malibu libraries. See the online help file, Malibu.chm or Malibu.hlp for details on all available classes, methods and events.

Malibu Library Features

Event Handlers

Often in a library it is useful to call user provided code at various points in the operation of the object. In traditional C, pointers to “callback” functions were often used for this purpose.

Malibu extends on the callback concept through use of events. In event-driven programming, user events are a key part of your application logic. An event is a mechanism that links an occurrence to some code. More specifically, an event is a closure that points to a method in a specific class instance.

From the application developer’s perspective, an event is just a name related to a system occurrence, such as OnProgress, to which specific code can be attached. For example, when an error is detected during a system operation, the OnError event might be called.

User events correspond to the elements in your application . For example, during a download operation, the OnDownloadComplete event is called by the system following delivery of the last portion of the executable image to the target DSP. This event can be programmed within application code to update the user interface in an appropriate manner. The code you write to respond to events is called an event handler.

Events and their user-written handlers (called closures) are a key part of the Malibu environment. They give us the ability to assign an event handler for system occurrences directly within application code.

The P25M in the Host Environment

The Malibu Library uses the supplied P25M device driver for downloading and communication via the PCI bus. The details of this are managed internally by the P25M class.

Connection to the Hardware
Each instance of the P25M class manages a single board. Systems containing multiple targets are supported by instantiating multiple objects. The boards are distinguished by a logical board number starting with zero. After construction, the Target() method may be used to declare which target the object will associate itself with.

At this point, the board object is not actually attached to the hardware. The Open() method establishes communication to the board assigned. If there is no hardware for that target, Open() will throw an exception. Once communications are completed, a call to Close() discontinues communications.

COFF loading to the 6713 Processor
One major requirement is to be able to download and run code on the 6713 over the network. The method DownloadCoff() within the Cpu() data member of the P25M object is used to begin a COFF load as a background operation. During the load process, the background thread automatically calls the baseboard class's OnDownloadProgress and OnDownloadComplete closures to provide interim status. DownloadCoff() merely posts a download request into a background thread that actually performs the download of the file passed as as an argument. This method does not block, and will return as soon as the request is posted.

Note that the target application may not be fully initialized and running when RequestCoffLoad() is finished, and even after the OnDownloadComplete closure is called, since the target initialization routines could be arbitrarily time-consuming . The best approach is to have the host application wait for some message to be sent from the target program which indicates that the target is fully prepared for further communication. The provided example programs show the use of this technique with the Login message.

Target / Host Packet Communication
Most applications require some form of data exchange between the target application and the host application during the running of the program. The P25M has a bi-directional data channel configured that can be used to send commands and bulk data between the Host CPU and the 'C6713 DSP.

The communication protocol on this channel supports sending special data blocks consisting of a small header plus an arbitrarily large data packet. These data packets are of type Innovative::PmcBuffer. The header within a PmcBuffer contains two words of information designated the PeripheralId and the PacketSize. The PeripheralId is an arbitrary tag value stored

into the header by the sender intended to allow the receiver to uniquely identify the purpose and contents of the packet. The PacketSize field is automatically updated by the driver to accurately reflect the size of the data portion of the packet, in 4-byte words. This field allows the receiver to determine the amount of data payload communicated within the packet.

The Send() method transfers both the bulk data and header stored in an Innovative::PmcBuffer object to the target.

Completion of a Send() requires that a thread within the target DSP application be blocking within a call to Transfer::Recv(). If the target has not done so or the system is otherwise unready, both the sender and receiver will block and not return until the data is completely transfered out and the buffer arguments can be reused. This makes these methods possibly unsafe to call from user-interface code in a Windows application. Since there is some internal queuing in the system, the completion of the send may not mean that the target has actually received and processed the packet, either. In general, it is best to call the functions Send and Recv from within background threads only on both Host and Target. However, the Malibu library implements a very efficient event (callback) scheme which can circumvent this requirement in some common cases. See the ASnap example for an illustration.

To receive data and commands from the target, use the Recv() method. This function will not return until the request is satisfied by receipt of a PmcBuffer from the target. The PeripheralId field within the header of this buffer may be examined in order to retrieve an application-specific code describing the nature of the data in the packet and the PacketSize field contains the size of the data payload contained in the buffer.

The code below illustrates the typical sequence used in the receipt of a buffer from the target:

```
//---------------------------------------------------------------------------
//  TForm1::HandleDataAvailable() --  Handler for all received target packets
//---------------------------------------------------------------------------

void TForm1::HandleDataAvailable(Innovative::PacketStreamDataEvent & Event)
{
    static PmcBuffer Packet;
    static int LoginTally(0);
    //
    // ...Get the packet from the system
    Event.Sender->Recv(Packet);
     //
    // ...Process the packet
    short PacketType = Packet.Header()->PeripheralId();
```

In this code, Packet is the PmcBuffer which will contain the data received from the target. The call to Event.Sender->Recv(Packet) blocks until a complete data packet has been received from the target. However, as illustrated in the ASnap example provided with the board, an asynchronous event notification mechanism is provided which provides a callback to a user application function such as HandleDataAvailable above when data is sent from the target to the host. In that context, the call to the Recv will complete immediately, since data has already arrived.

Once the packet is received, it is common practice to examine the header to discover the type of message sent, then dispatch accordingly. The call to pda.Header()->PeripheralId() reads the ID tag sent by the target from the PmcBuffer header. Ordinarily, the application uses a switch table to execute the appropriate code for each ID code that can be received. For example:

```
    switch (PacketType)
        {
        case ccAcqusition:
            {
            Report->Line = "pmAcqusition";
            ProgressBar->StepIt();

            StatusBar->Panels->Items[1]->Text = "Rx: " + String(++RcvMessageTally);
```

```
            Cache->Write(Packet.IntPtr(), Packet.Size()*sizeof(int));
            }
            break;

        ...
```

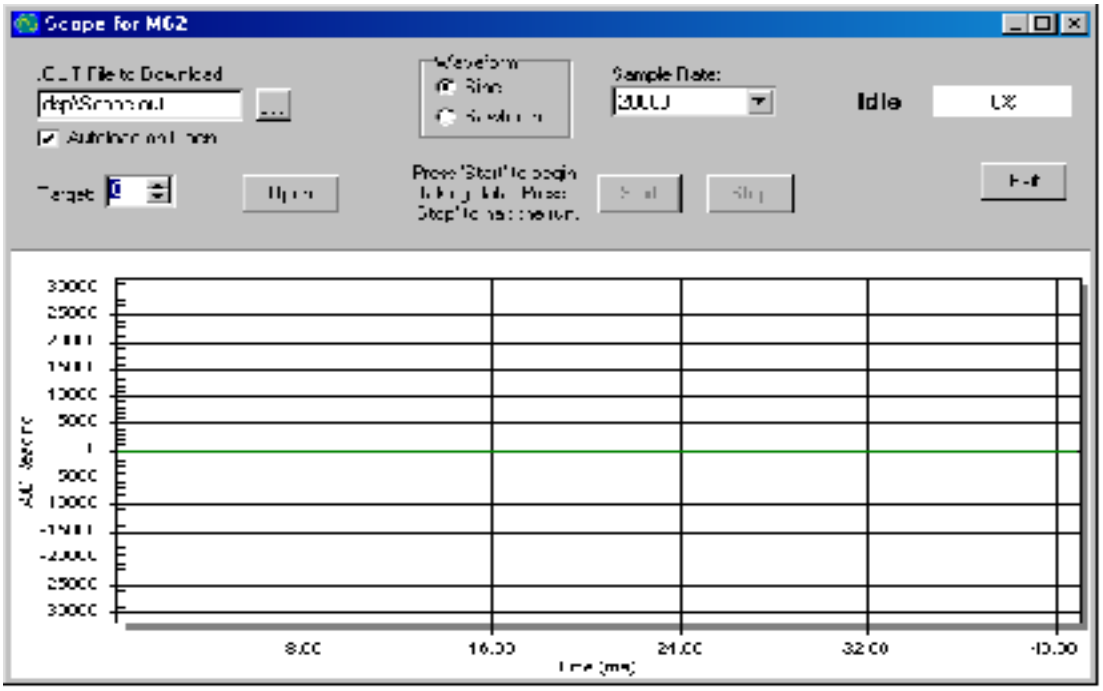
When the `ccAcqusition` code is received, this application logs the contents of the data payload in the received buffer to disk. The call to `Cache->Write()` calls the Host OS to perform a disk write from the data buffer starting at address `Packet.IntPtr()` which is the address of the first byte of the data payload within the packet, cast as an integer pointer. The number of bytes in the payload are returned by the phrase `pda.Size()*sizeof(int)`.

The target software uses a similar method for the other end of the connection. The **Transfer** class has identical *Send(),* and *Recv()* methods of its own for the same purposes.

Scope Host Program

Each Zuma Toolset is supplied with one or more example programs, which illustrates control of the DSP board via the supplied DLL. Most DSP products include the example, SCOPE, which emulates a simple oscilloscope. This example is detailed below.

The SCOPE example is a small, working example written in Borland C++ (Host side) showing how to use bus-based DSP boards to move data between the target and Host memory spaces. The host application works in concert with a small DSP program (written in Texas Instruments C) running on the target to mimic the operation of a simple oscilloscope. This program illustrates many of the elements of a typical Host application, which communicates with a target DSP application using the Zuma DLL functions.



The SCOPE application is included in the \EXAMPLES\HOST\SCOPE subdirectory of %II_BOARD%. The DSP support code for this application is located in the \EXAMPLES\HOST\SCOPE\DSP subdirectory.

All of the Host user-interface code has been placed into the ScopeMain.cpp file. All of the DSP control and DLL interface functions have been factored into the DspBoardFtns.cpp file.

Concentrate on understanding the code in the DspBoardFtns.cpp file, since this file contains the substance of the example program - downloading code to the target DSP; communicating commands and data between the target and host processor, etc. This file utilizes all of the most commonly-used DLL functions needed in typical application programs involving Host application control of a target DSP board. The contents of this file are minimally dialect-dependent, so that porting this code to Visual C++ or another language should be straightforward.

The ScopeMain.cpp file contains C++ code, which is specific to the Borland C++ dialect. This code will not readily port to other dialects or languages since the Borland RAD (rapid application environment) automatically takes care of nearly all of the Windows programming details which are often visible and burdensome in other C++ implementations (such as Visual C++).

DspBoardFtns.cpp Functions
int InitScope(int target, bool autoload, char * OutFile);

The InitScope function opens the target DLL and downloads the target application to the DSP board. The function synchronizes with the target, using mailbox input, including a timeout to accommodate the case where the download fails or the target is not viable for some reason. The ScopeISR() function is installed and enabled as the virtual interrupt service routine (VISR) for the host application. The VISR must be installed and enabled before the target application will be able to

asynchronously interrupt the host application to indicate arrival of new analog input data. The address of the host bus-master memory pool is sent to the target application, via a mailbox, followed by a final synchronization word.

void StartScope(int target, ScopeWaveform WaveType, int SampleRate)

The StartScope function clears the internal data queue utilized by the VISR to contain data blocks received from the target during data acquisition. Additionally, the function sends a start acquisition command to the target DSP, using the write_mailbox() and mailbox_interrupt DLL functions, causing the target application to begin data acquisition and asynchronous transmission of acquired data to the host application. Note that the target application utilizes a simple command interpreter to dispatch commands (such as START_COMMAND in this example) received via the mailbox from the host.

void StopScope(int target);

The StopScope function simple sends the stop command (STOP_COMMAND) to the command interpreter on the target DSP, causing the target application to discontinue data acquisition and asynchronous transmission of acquired data to the host application.

void CloseScope(int target);

The CloseScope function disables the VISR and de-installs the interrupt handler. This should be accomplished prior to attempting to change the VISR handler or terminating the application. This function also closes the DLL, which relinquishes internal DLL resources and allows other applications to utilize the DLL, if needed.

```
void  ScopeISR( void * context );
```

The ScopeISR function is executed in response to asynchronous data available messages from the target DSP. When the target has accumulated a packet of data from the mock analog input device (interrupt- driven) it sends the data to the host via the bus interface and issues a mailbox_interrupt() call to notify the host of data availability.

The DLL receives notification of this event from the device driver and calls the function previously installed using the `host_interrupt_install()` DLL function - `ScopeISR()`.

ScopeISR reads the interrupt mailbox to determine which half of the bus-master memory pool has been updated by the target DSP with newly acquired data. The function copies this data into a queue, imaginatively names Queue, which the mainline code can access in order to process and plot on the graphical display.

```
int  ScopeProcess();
```

The ScopeProcess function is called by the user interface code in order to determine the fullness level of the VISR-maintained data queue, so that the GUI display can be periodically updated.

```
int  Download( int target, char * OutFile );
```

The Download function embodies all of the functionality needed to download a target DSP application and cause it to begin executing. This code is usually performed automatically in user applications as the host application is invoked. In the SCOPE example, the target code is downloaded in response to a button click, to emphasize the procedure.

When the Host program starts, it invokes the COFF downloader built into the DLL to download the object image of the target DSP application (SCOPE.OUT) within the Download() procedure. This procedure makes calls on the DLL in order to effect the download. Following the download, the application is started running using the start_app() function. Upon receipt of the

START_COMMAND, the DSP application begins generating mock analog data in order to emulate acquiring data from the analog subsection on the target, enqueuing the acquired data in a target-maintained data queue. As soon as a packet-full of data is available, the data is dequeued by the target, moved into shared memory and the Host program is signaled, using the host_interrupt() target procedure.

The Host device driver handles the target interrupt signal and issues an special EVENT message to the ring three DLL which performs a callback on the user-installed Host ScopeISR() function. When this occurs, the offset into shared memory containing the new packet of analog samples is read into the Host queue, Queue.

The Host user-interface code in ScopeMain.cpp draws an oscilloscope-like grid on the display window, then polls continuously for the availability of analog samples in the Host queue. When a screen full of data is available in the queue, it is dequeued and plotted.

Host Example Programs for the P25M Baseboard

Overview

Command and Data Packets

On this baseboard, using PCI bus, the way to transport information is by means of asynchronous packets of data that are transferred and decoded by the destination. These messages may be of varying sizes, allowing large messages to be efficient in sending bulk data while allowing small command messages to be mixed into the data stream. When delivered to the destination, the messages can be parsed and can result in any kind of processing desired.

Unlike some other baseboards, the P25M uses PmcBuffers for both command and data, since there is only a single communications channel. This makes the distinction between the tow types of packets somewhat arbitrary. Usually command packets affect program control flow, with a relatively modest number of argument data. Data packets usually are large and hold bulk information.

By having the receipt of a command packet trigger the generation of additional messages, a protocol can be developed to allow the transfer of data or the execution of control functions on demand from the other side of the link. This is even more natural if the applications are written in an event-driven style. The arrival of messages are the events to which the application responds.

ASnap

The ASnap example is communications demonstration that resides in the \P25M\Examples\ASnap directory. This example demonstrates continuous, high-rate analog acquisition from a suitable Omnibus module into a buffer on the target DSP. After the acquisition, the data is sent to a Windows disk file via the PCI interface. This data may be examined using the supplied BinView applet. Simple bidirectional communications between the target and the host are shown.

The target project and its source are located in the \P25M\Examples\Asnap directory. The host project is located in the \Vc7 subdirectory beneath the target project. This contains the source for the Microsoft MSVC version of the Host example. The Borland BCB version of the Host example is available on request.

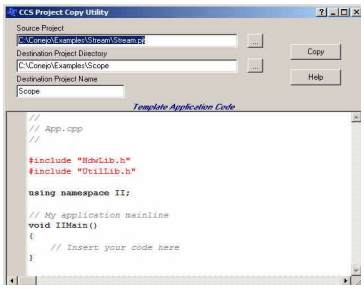
DataXfer

The DataXfer example is communications demonstration. It demonstrates a two way flow of communication using a protocol of command packets and data packets.

The target project and its source are located in the \P25M\Examples\Asnap directory. The host project is located in the \Vc7 subdirectory beneath the target project. This contains the source for the Microsoft MSVC version of the Host example. The Borland BCB version of the Host example is available on request.

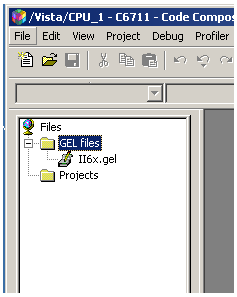
Building a Target DSP Project

Building a project suitable for a Matador or Velocia baseboard requires a particular setup of the project.

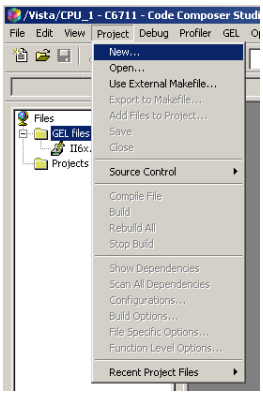


By far, the easiest way to create a new DSP project is by using an existing project as a template. The CopyCcsProject applet provided in the Pismo Toolset automates this task. To use this utility, select an existing Code Composer project as the Source Project, typically one of the example programs supplied in the Pismo Toolset. Next, select the directory into which you wish the new project to be created, using the Destination Project Directory edit control. Then, edit the Destination Project Name for the newly-created project. Finally, click the Copy button to create the new project from the template. The new project may be opened and used within Code Composer.

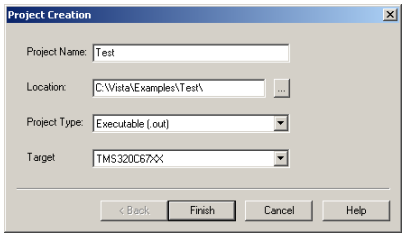
Alternately, you may follow the manual steps below to create a new target DSP project. The project name used below is called Test, but you should name your project appropriately for your application.



Start Code Composer Studio. In the default configuration, the project window will contain no projects but will contain the default Innovative-supplied board initialization GEL file.

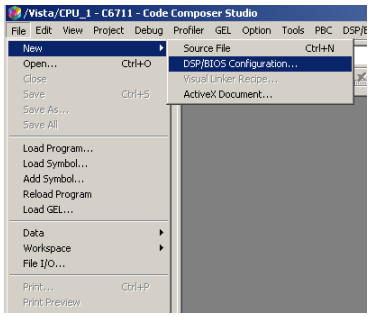


Click Project | New on the menu bar to create a new DSP project.

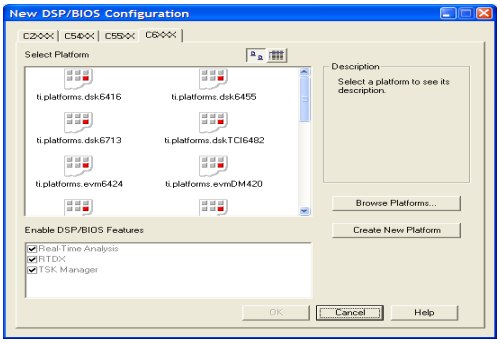


Specify the location for the new project and its name. In this example, a new project called Test is being created in the `Sbc6711-Pismo\\Examples\\` directory. Change the location to accommodate your board type and processor type.

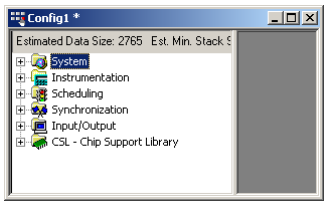
After the new project has been created, it will appear in the CCS project window under the Projects folder.



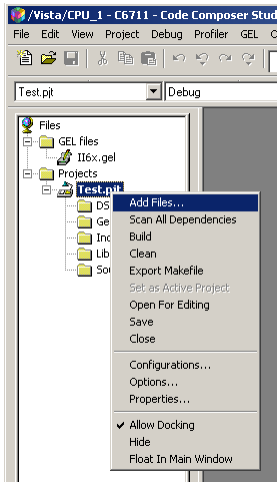
Click File | New | DSP/BIOS Configuration to create a new TCF file for use in the project.



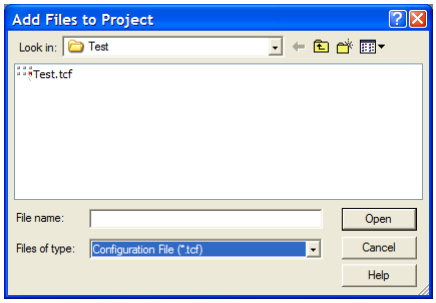
Select the relevant template for the baseboard from the list in the New DSP/BIOS Configuration dialog box.



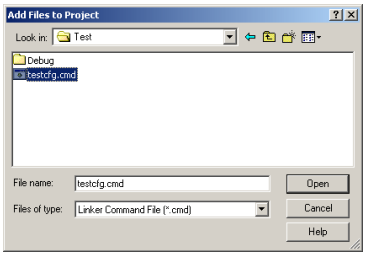
By default, this TCF will be named Configuration1. Save it as `Test.TCF`.



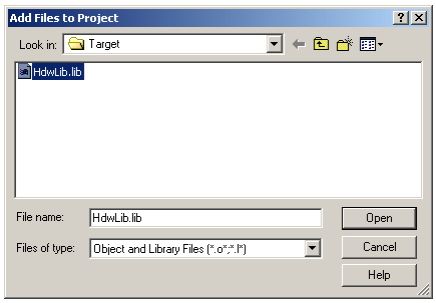
Though the TCF and its support files have been created on disk, you must manually add them to the Test project. Right-click on `Test.pjt` in the Project window to invoke the project hot menu. Click Add Files to add a file to the project.



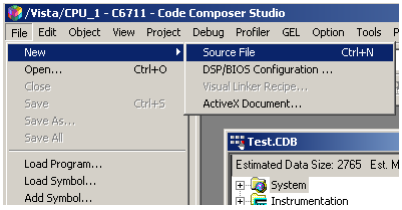
Select the the newly-created `Test.tcf` for addition to the project. This will implicitly add the auto-generated files `Testcfg.s62 (Testcfg.s64 for Velocia cards)` and `Testcfg_c.c` to the project as well.



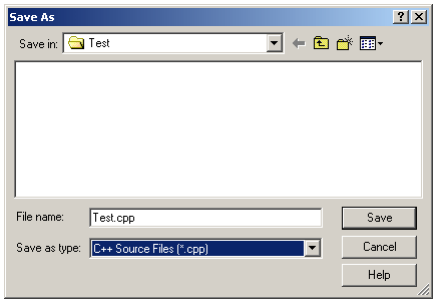
Right-click on `Test.pjt` in the project window, click “Add Files” then select the the newly-created `Test.cmd` for addition to the project.



Right-click on `Test.pjt` in the project window, select Add Files, then browse to the Examples directory and select `Examples.cmd` for addition to the project.



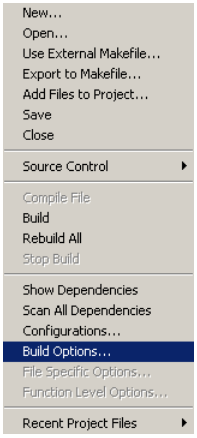
Add an new C++ source file to the project. Click File | New | Source File to create an empty source document.

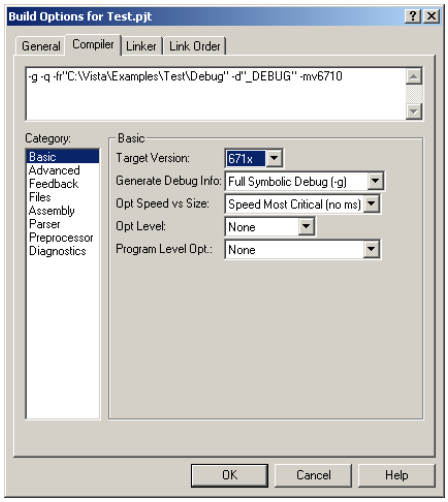


Rename the new source document to `Test.cpp`. To use the Pismo libraries, you must use C++ files and the C++ compiler, even if you intend to restrict your own coding to the C subset of C++

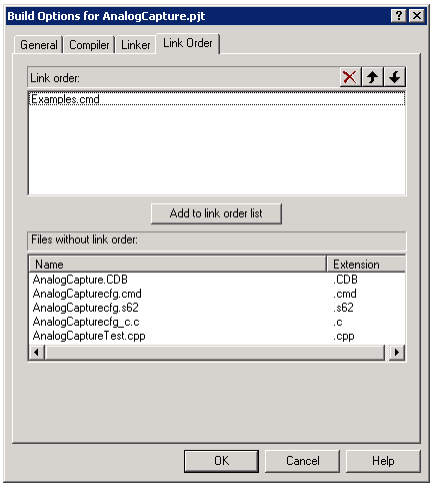
```
//
// App.cpp
//

#include "Pismo.h"

using namespace II;

// Application mainline
void IIMain()
{
    // Insert your code here
}
```

Type the boilerplate code below into your source file. This is the minimum code needed for any Pismo C++ application.

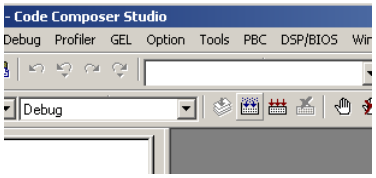




Click the menu Project | Build Options to invoke the compiler Build Options dialog. Then select the Files Category, then enter the pathspec to the `Examples.opt` file in the Examples directory to the Options File edit box.



Click on the Link Order tab, then add `Examples.cmd` to the Link Order List.



Click the Incremental Build button to rebuild the template application. It should compile and link without errors.

Writing a Program

The basic program given in the example above includes a 'Main' function `IIMain()`. DSP/BIOS, the OS used in the Pismo library, uses code inserted after exiting from the normal C-language `main()` to initialize features of DSP/BIOS. This means that some language features are not available then. To avoid these problems the Pismo library provides a `main()` function and uses it to create a single thread. This thread, when executed, calls the `IIMain()` function. Inside of this thread, all DSP/BIOS is initialized and ready for use. It is required that the user include this function, and use it as the equivalent of the old main process function in C.

Host Tools for Target Application Development

The Innovative Integration Pismo Toolset allows users of Innovative DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C++ Compiler, Assembler and Linker, the Code Composer debugger and code authoring environment as well as Innovative's custom Windows applets (such as the terminal emulator).

Code Composer Studio is the package used to automate executable build operations within Innovative's Pismo Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within Code Composer Studio, then downloaded to the target and tested within either the Code Composer Studio debugger or via the terminal emulator.

Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms are given in the above section of this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turnkey application with custom target and host software, *two* programs must be written for two separate compilers. While Innovative supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C++ language. For information on C/C++ language basics, consult one of the primer books available at your local bookstore.

Components of Target Code (.cpp, .tcf, .cmd, .pjt)

In general, DSP applications written in TI C++ require at least three files: a .cpp file (or "source" file) containing the C++ source code for the application a .cmd file (or "command" file) which contains the target-specific memory-map and build data needed by the linker, a .tcf file (or "command database" file) which specifies the properties of the BIOS operating system used within the application and a .pjt file ("project" file) which centralizes all project-specific options, settings and files. There may also be one or more .asm assembler source files, if the user has coded any portions of the application in assembly language.

Edit-Compile-Test Cycle using Code Composer Studio

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion. In the Innovative Integration development system these stages are accomplished within the Code Composer integrated development environment (IDE).

By using Code Composer Studio, these stages of the programming cycle are accomplished entirely within the IDE. The project features of Code Composer Studio support component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware. This fully integrated programmers environment is more user-friendly then the basic command line interface, which comes standard with the TI tools.

Automatic projectfile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.pjt`, located in the project directory), which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Code Composer Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select Project | Rebuild All from the Code Composer Studio menu bar.

IIMain replaces main.
Due to restrictions within Dsp/Bios, not all BIOS features may be safely used within `main()`, since it is called early in the system initialization sequence. To circumvent this limitation, Pismo automatically constructs a default thread running within normal priority and starts this thread automatically. The entry point function in this thread is called IIMain, and all Pismo applications must define this function. This function is intended to replace main in your application programs. You may safely call any BIOS function within IIMain.

Running the Target Executable

The test program may be converted into a simple, "Hello World!" example, by using the built-in standard I/O features within Pismo. Bring up the `Test.cpp` source file edit screen. Scroll down the source file by using cursor down button until you reach the `IIMain()` function. Edit it as follows:

```
#include "HdwLib.h"
#include "UtilLib.h"

        cio << init;
cio << "Hello World!" << endl;
```

```
cio.monitor();
```

You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Code Composer Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`test.obj`). Code Composer Studio then invokes the TI Linker using the `testcfg.cmd` file, which is located in the project directory. This rebuilds the executable file using the newly revised `test.obj` . If no errors were encountered, this process creates the downloadable COFF file `test.out`, which can be run on the target board. At this point, the program may be run using the terminal emulator applet, which may be invoked using the terminal emulator shortcut located within the target board program group created during the Pismo Libraries installation process. In the terminal emulator, download the `test.out` file. The program runs and outputs the message "Hello, World" to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (such as a C++ syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the build output window.

Note:
Be sure to start the terminal emulator **BEFORE** starting Code Composer, to avoid resetting the DSP target in the midst of the debugging session. If the terminal emulator is not yet running and you wish to run the Test object file, perform the following steps.

1. Execute Debug | Run Free to logically disconnect the DSP from the debugger software.

2. Terminate the Code Composer Studio application.

3. Invoke the terminal emulator application.

4. Restart the Code Composer Studio application.

This outlines the basics of how to recompile the existing sample programs within the Code Composer Studio environment.

Anatomy of a Target Program

While not providing much in the way of functionality, the `test` program does demonstrate the code sequence necessary to properly initialization the target. The exact coding, however, is very specific to the I.I. C Development Environment, target boards, and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

```
/*
*       HELLO.CPP
*       Test file/program for target board.
*/

#include "Pismo.h"
```

```
IIMain()
{
        cio << init;

        cio << "Hello World!" << endl;
        cio << "\nEchoing keystrokes..." << endl;

        char key;

        do
                {
                cio >> key;
                cio << key << flush;
                }
        while(key != 0x1b);

        cio.monitor();
}
```

The two lines of the program that being with a "#" are #include statements, which include the header files for the hardware and utility I/O libraries. These include prototypes for all the library classes within Pismo.

The `cio << init` invocation will setup the standard monitor I/O interface and reset the terminal window. The next lines perform the basic standard I/O functions of printing "Hello World!" & "Echoing keystrokes...". These two lines are where custom code could be inserted.

The following do-loop sequence simply echoes keys typed at the terminal emulator back to the terminal display, until the `Esc` key is pressed. When `Esc` is pressed, the `cio.monitor()` function effectively terminates the program, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `test` program is very simple, but it contains the basic components of a typical DSP application, as well as the initialization needed to interact with the terminal emulator.

Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference within the Pismo online help for library location information on each class and method.

In general, user software needs to #include the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The files `HdwLib.h` and `UtilLib.h` should be included within all programs; The file `DspLib.h` should be included if a program uses functions in the DspLib signal processing library.

Example Programs

Under **<baseboard>\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under

Borland Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.3 and DSP/BIOS. Note that not all of the examples listed below are available for all targets.

Table 30. Pismo Example Programs

Example	Host	Target	Illustrates
FftFix	terminal emulator	DSP/BIOS	Use of Fourier class to perform forward and inverse FFTs
FirFix	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
CpuInRate	BCB MSVC	DSP/BIOS	Use of Target to Host message and data packet passing via PCI bus.
CpuOutRate	BCB MSVC	DSP/BIOS	Use of Host to Target message and data packet passing via PCI bus.
LinkPort	BCB	DSP/BIOS	Use of LinkPort driver to flow data between all processor in mesh
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Developing Host Applications

Developing an application will more than likely involve using an integrated development environment (IDE) , also known as an integrated design environment or an integrated debugging environment. This is a type of computer software that assists computer programmers in developing software.

The following sections will aid in the initial set-up of these applications in describing what needs to be set in Project Options or Project Properties.

Borland Turbo C++

BCB10 (Borland Turbo C++) Project Settings

When creating a new application with File, New, VCL Forms Application - C++ Builder

Change the Project Options for the Compiler:

Project Options
++ Compiler (bcc32)
C++ Compatibility
Check 'zero-length empty base class (-Ve)'
Check 'zero-length empty class member functions (-Vx)'

In our example Host Applications, if not checked an access violation will occur when attempting to enter any event function.

i.e.
Access Violation OnLoadMsg.Execute – Load Message Event
Because of statement
Board->OnLoadMsg.SetEvent(this, &ApplicationIo::DoLoadMsg);

Change the Project Options for the Linker:

Project Options
Linker (ilink32)
Linking – **uncheck 'Use Dynamic RTL'**

In our example Host Applications, if not unchecked, this will cause the execution to fail before the Form is constructed.

Error: First chance exception at $xxxxxxxx. Exception class EAccessViolation with message "Access Violation!"
Process ???.exe (nnnn)

Other considerations:

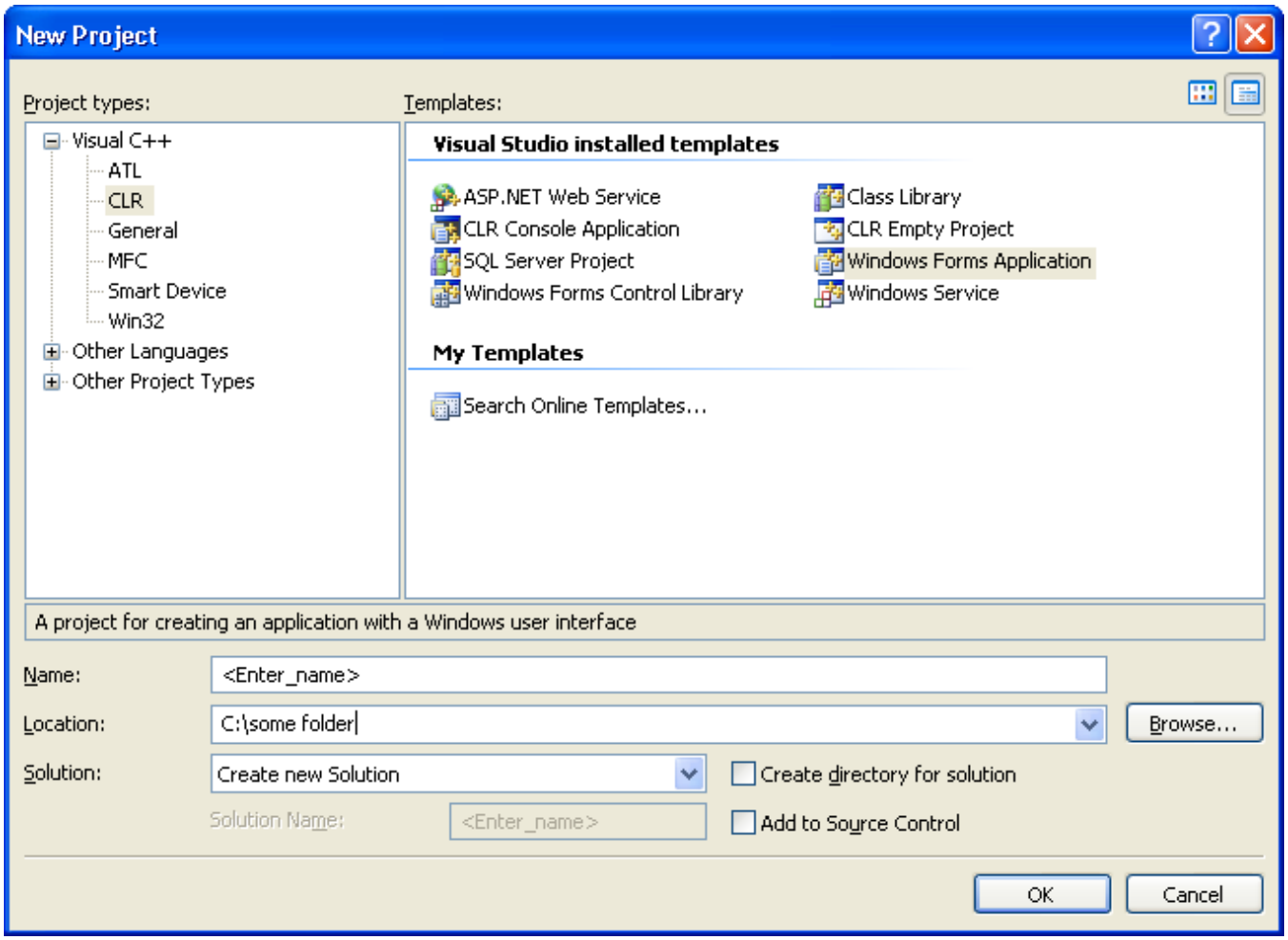
Project Options
++ Compiler (bcc32)
Output Settings
check – Specify output directory for object files(-n)
(release build) Release
(debug build) Debug
Paths and Defines
add Malibu
Pre-compiled headers
uncheck everything
Linker (ilink32)
Output Settings
check – Final output directory
(release build) Release
(debug build) Debug
Paths and Defines
(ensure that Build Configuration is set to All Configurations)
add Lib/Bcb10
(change Build Configuration to Release Build)
add lib\bcb10\release
(change Build Configuration to Debug Build)
add lib\bcb10\debug
(change Build Configuration back to All Configurations)

Packages
uncheck - Build with runtime packages

Microsoft Visual Studio 2005

Microsoft Visual C++ 2005 (version 8) Project Properties

When creating a new application with File, New, Project with Widows Forms Application:



Project Properties (Alt+F7)
Configuration Properties

Project Defaults	
Configuration Type	Application (.exe)
Use of MFC	Use Standard Windows Libraries
Use of ATL	Not Using ATL
Minimize CRT Use in ATL	No
Character Set	Use Unicode Character Set
Common Language Runtime support	Common Language Runtime Support (/clr)
Whole Program Optimization	No Whole Program Optimization

C++
General
Additional Include Directories

Malibu
PlotLab/Include – for graph/scope display

Code Generation
Run Time Library

Multi-threaded Debug DLL (/Mdd)

Precompiled Headers
Create/Use Precompile Headers

Not Using Precompiled Headers

Linker
Additional Library Directories

Innovative\Lib\Vc8

If anything appears to be missing, view any of the example sample code Vc8 projects.

DialogBlocks

DialogBLocks Project Settings (under Linux)

Project Options
[Configurations]
Compiler name = GCC
Build mode = Debug
Unicode mode = ANSI
Shared mode = Static
Modularity = Modular
GUI mode = GUI
Toolkit = <your choice wxX11, wxGTK+2, etc>
Runtime linking = Static or Dynamic, we use Static to facilitate execution of programs out of the box.
Use exceptions = Yes
Use ODBC = No
Use OpenGL = No
Use wx-config = Yes
Use insalled wxWidgets = Yes
Enable universal binaries = No
...
Debug flags = -ggdb -DLINUX
Library path = %INNOVATIVE%/Lib/Gcc/Debug, %WINDRIVER%/lib
Linker flags = %AUTO% -Wl, @%PROJECTDIR%/Example.lcf
IncludePath= -I%INNOVATIVE%/Malibu -I%INNOVATIVE%/Malibu/LinuxSupport %AUTO%

[Paths]
INNOVATIVE= /usr/Innovative
WINDRIVER= /usr/Innovative/WinDriver
WXWIN= /usr/wxWidgets-2.8-7 provided that this is the location where you have installed wxWidgets.

Summary

Developing Host and target applications utilizing Innovative DSP products is straightforward when armed with the appropriate development tools and information.

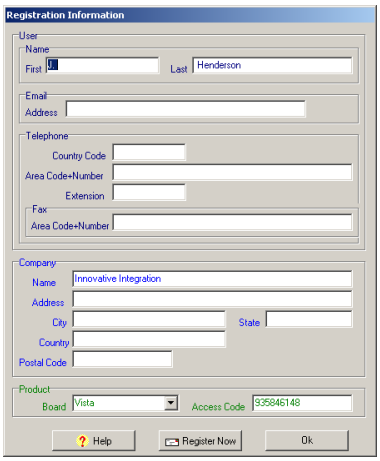
Applets

The software release for a baseboard contains programs in addition to the example projects. These are collectively called "applets". They provide a variety of services ranging from post analysis of acquired data to loading programs and logic to a full replacement host user interface. The applets provided with this release are described in this chapter.

Shortcuts to these utilities are installed in Windows by the installation. To invoke any of these utilities, go to the Start Menu | Programs | <<Baseboard Name>> and double-click the shortcut for the program you are interested in running.

Registration Utility (NewUser.exe)

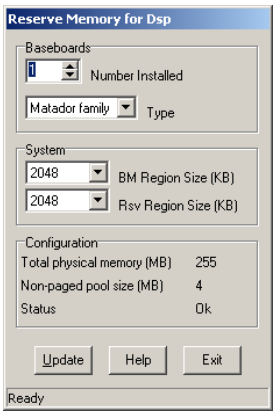
Some of the Host applets provided in the Developers Package are keyed to allow Innovative to obtain end-user contact information. These utilities allow unrestricted use during a 20 day trial period, after which you are required to register your package with Innovative. After the trial period operation will be disallowed until the unlock code provided as part of the registration is entered into the applet. After using the NewUser.exe applet to provide Innovative Integration with your registration information, you will receive:



The unlock code necessary for unrestricted use of the Host applets

A WSC (tech-support service code) enabling free software maintenance downloads of development kit software and telephone technical hot line support for a one year period.

ReserveMemoryDsp



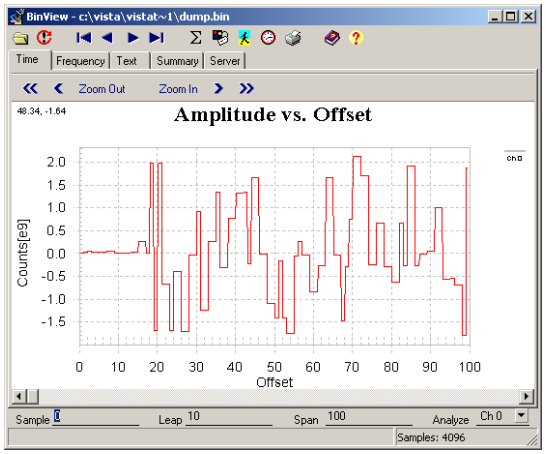
Each Innovative PCI-based DSP baseboard requires from 2 to 8 MB of memory to be reserved for its use, depending on the rates of bus-master transfer traffic which each baseboard will generate. Applications operating at transfer rates in excess of 20 MB/sec should reserve additional, contiguous busmaster memory to ensure gap-free data acquisition.

To reserve this memory, the registry must be updated using the ReserveMemDsp applet. If at any time you change the number of or rearrange the baseboards in your system, then you must invoke this applet to reserve the proper space for the busmaster region. See the Help file ReserveMemDsp.hlp, for operational details.

Data Analysis Applets

Binary File Viewer Utility (BinView.exe)

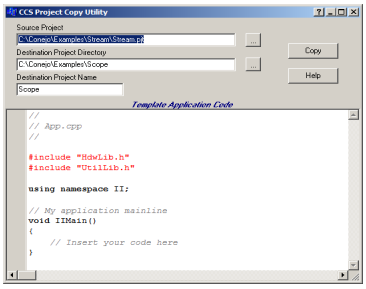
BinView is a data display tool specifically designed to allow simplified viewing of binary data stored in data files or resident in shared DSP memory. Please see the on-line BinView help file in your Binview installation directory.



Target Programming Applets

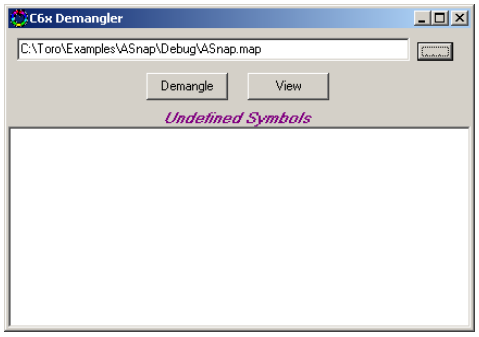
Target Project Copy Utility (CcsProjectCopy.exe)

The CopyCcsProject.exe applet is used to copy all project settings from a known-good template project into a new DSP Code Composer project. This simplifies new project development, by eliminating the multi-step process of copying the myriad individual project settings from a source project in a newly-created project.



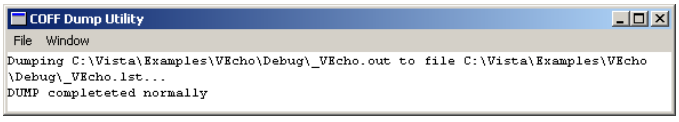
Demangle Utility (Demangle.exe)

The Demangle applet is designed to simplify use of the TI dem6x.exe command-line utility. When building C++ applications, the built-in symbol mangler in the TI compiler renders symbolic names unreadable, such that missing or unresolved symbol errors displayed by the linker no longer correlate to the symbol names within your code. To work around this limitation, enable map file generation within your CCS project. Then, browse to the map file produced by the linker using the Demangle utility. The utility will display proper symbol names for all unresolved externals.



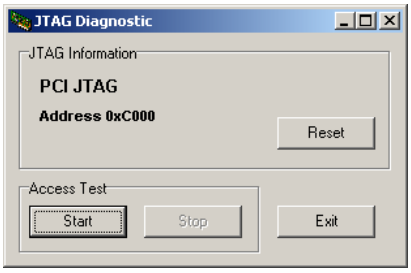
COFF Section Dump Utility (CoffDump.exe)

CoffDump.exe parses through a user-selected COFF file stored on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer.



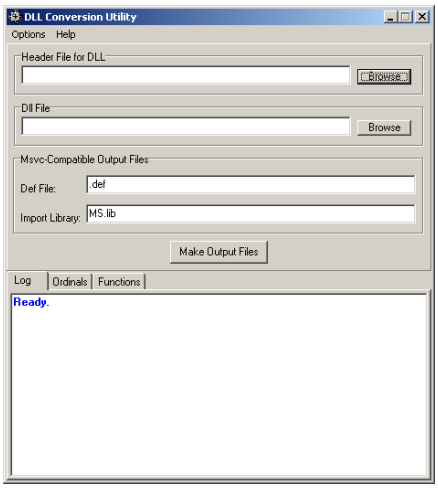
JTAG Diagnostic Utility (JtagDiag.exe)

JtagDiag.exe is used to re-initialized the JTAG scan-path interface which connects the Code Hammer debugger's PCI plug-in board with the target DSP. Use this utility prior to invoking Code Composer Studio, to insure that the communications link is viable and clear. This utility is also convenient in confirming that the Code Hammer installation is complete and correct.



DEF Conversion Utility (DefConvert.exe)

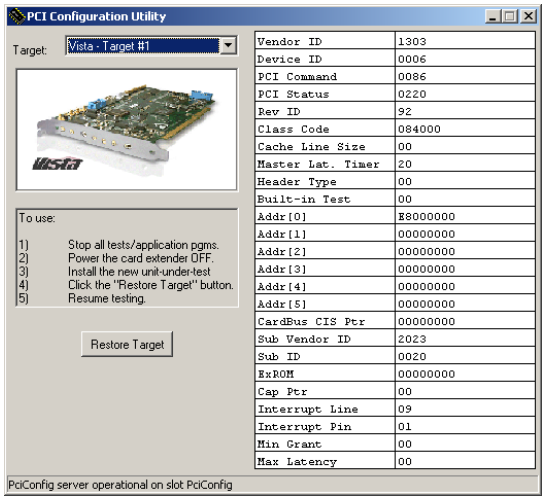
The DefConvert.exe applet is a simple utility to aid in the use of DLLs written using Borland Builder from within Microsoft Visual C/C++- application programs. DefConvert parses a user specified C++ header file and DLL in order to create a MSVC compatible DEF file. This DEF file is then submitted to the MSVC LIB utility in order to generate an MSVC-compatible import library (.LIB).



Complete functionality is supplied in the DefConvert.hlp file.

PCI Configuration Utility (Config.exe)

The Config.exe applet is used to view all of the PCI configuration space variable data for any of Innovative Integration's PCI baseboards. This tool is generally used as a low-level diagnostic to troubleshoot installation problems. But it is also capable of restoring previously-memorized PCI config-space enumeration settings in the instance of target application crash, which inadvertently disrupts PCI config information.

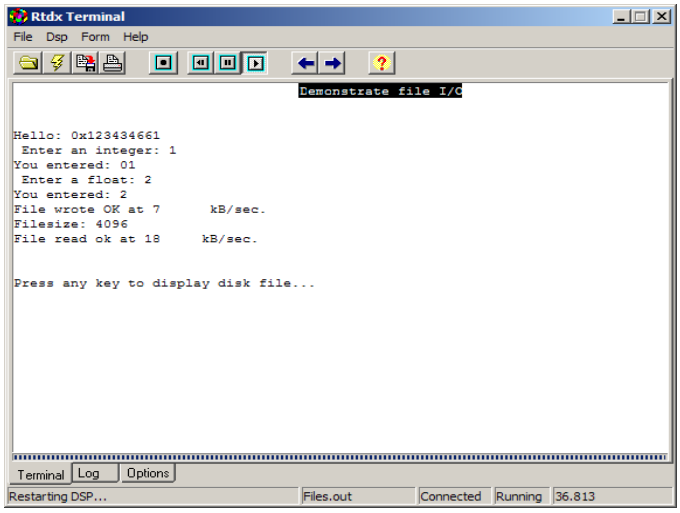


RtdxTerminal - Terminal Emulator

This applet provides a C++ language-compatible, standard I/O terminal emulation facility for interacting with the TermIo library running on an Innovative Integration target DSP processor. Display data is routed between the DSP target and this Host the terminal emulator applet in which ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation during initial debugging.

RtdxTerminal is implemented as an out-of-process extension to Code Composer Studio. Consequently, it must be used in conjunction with CCS and a JTAG debugger - it cannot operate stand-alone.

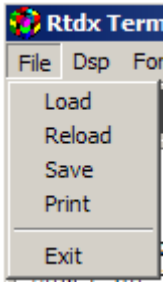
The terminal emulator is straightforward to use. The terminal emulator will respond to stdio calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. The DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking. The stdio output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.



The terminal emulator also supports Windows file I/O using the TermFile library object.

Important Note:
Before using the terminal emulator, you must register your Pismo Toolset. Until you do so, usage will be restricted to a 20-day trial period for the terminal emulator and other applets contained in the Toolset. To register, fill out the contents of the Registration Form, then click on the Register Now button. This will print a Registration report which, must be faxed to Innovative Integration. Innovative Integration will E-mail you an Access Code, which must be typed into the Registration Form for all the features to be enabled.

Terminal Emulator Menu Commands
The terminal emulator provides several menus of commands for controlling and customizing its functionality. These functions are available on the menu bar, located at the top of the the terminal emulator main window. Speed button equivalents for each of the menu options are also available on the button bar located immediately beneath the menu bar. The following is a description of each menu entry available in the terminal emulator, and its effects.



The File Menu
File | Load - provides for COFF (Common Object File Format) program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the full pathname to the COFF filename to be downloaded is selected by the user. Clicking "Open" in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking "Cancel" will abort the file selection and close the requester with no download taking place.

This operation can optionally be initiated via the button.

File | Reload - Reloads and executes the COFF file last downloaded to the target. It provides a fast means to re-execute the application program most recently loaded into the target board.

This operation can optionally be initiated via the button.

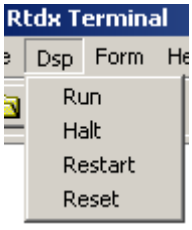
NOTE: File | Load and File | Reload functions use the JTAG debugger and Code Composer Studio in order to effect the program download.

File | Save – saves the textual contents of the Terminal and Log tabs to a user specified file.

File | Print - prints the textual contents of the Terminal and Log tabs to a user specified printer.

File | Exit – closes the emulator application, terminating console emulation.

The DSP Menu
Dsp | Run - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Run within Code Composer Studio.



This operation can optionally be initiated via the button.

Dsp | Halt - causes the terminal emulator to suspend DSP program execution. This is functionally identical to performing Debug | Halt within Code Composer Studio.

This operation can optionally be initiated via the button.

Dsp | Restart - rewinds the DSP program counter to the application entry point, usually *c_int00()*. This is functionally identical to performing Debug | Restart within Code Composer Studio.

This operation can optionally be initiated via the button.

Dsp | Reset - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Reset Dsp within Code Composer Studio.

This operation can optionally be initiated via the button.

The Form Menu

Form | Tuck Left - repositions the main application window to the bottom left of the Windows desktop.

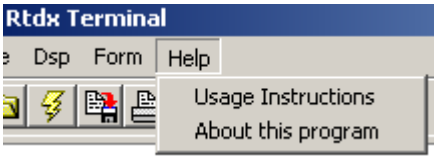
This operation can optionally be initiated via the button.

Form | Tuck Right - repositions the main application window to the bottom right of the Windows desktop.

This operation can optionally be initiated via the button.

The Help Menu

Help | Usage Instructions - displays online help detailing use of the application, including command-line arguments.



This operation can optionally be initiated via the button.

Help | About this Program - displays a dialog containing program revision and tech support contact information.

Options Tab:

The Options tab (seen below) contains controls to allow user-customization of the appearance and operation of the terminal emulator.

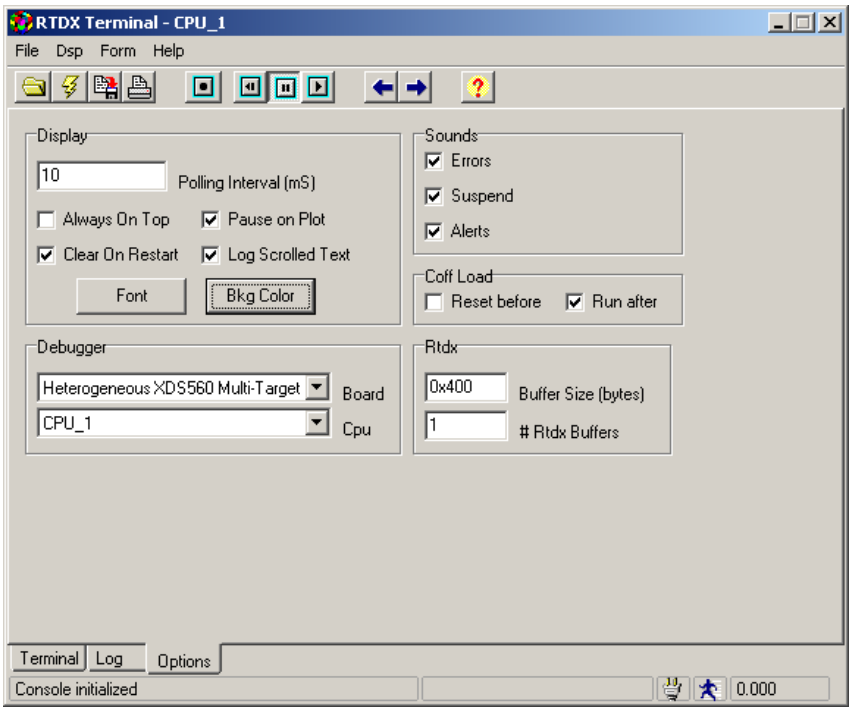


Figure 20. RtdxTerminal Options

Display *Group*
Controls within the Display group box govern the visual appearance of the terminal emulator, as detailed below.

Polling Interval - specifies the period, in milliseconds, between queries for data received from the DSP via the JTAG RTDX interface. Lower numbers increase performance but increase Host CPU load.

Always on Top - specifies that the terminal application should always remain visible, atop other applications on the Windows desktop. This check box controls whether the terminal emulator is forced to remain a foreground application, even when it loses keyboard focus. This is useful when running stdio-based code from within the Code Composer environment, when it's preferable to make terminal visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.

Clear on Restart - specifies whether the terminal display and log will be automatically cleared whenever the DSP is restarted.

Pause on Plot - specifies whether standard I/O will be suspended following display of graphical information in the Binview applet which is automatically invoked via use of the Pismo library

Plot() command. If enabled, standard I/O may be resumed by clicking the button.

Log Scrolled Text - specifies whether text information which scrolls off screen on the Terminal tab is appended to the Log display. If enabled, standard I/O performance will degrade slightly during lengthy text outputs.

Font - button invokes a font-selection dialog which allows selection of user-specified font within the Terminal and Log text controls.

Bkg Color - button invokes a color-selection dialog which allows selection of user-specified background color within the Terminal and Log text controls.

Sounds Group
Controls within the Sounds group box govern the audible prompts generated by the terminal emulator, as detailed below.

Errors - if enabled, file I/O and other errors encountered during operation generate an audible tone.

Suspend - if enabled, suspension of standard I/O, such as following plotting via Binview, generate an audible tone.

Alerts - if enabled, alert conditions encountered during standard I/O, such as upon display of the ASCII bell character, generate an audible tone.

Coff Load Group
Controls within the Coff Load group box govern behavior surrounding a COFF executable download.

Reset Before - if enabled, the Code Composer Debug | Reset DSP behavior is executed before attempting to download the user-specified COFF file.

Run After - if enabled, the Code Composer Debug | Run behavior is executed immediately following the download of a user-specified COFF file.

Debugger Group
Controls within the Debugger group box specify the target DSP with which RTDX communications is established.

Board - specifies the board hosting the target DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available board types configured using the Code Composer Setup utility.

Cpu - specifies the identifier of the specific DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available CPUs present on the baseboard as configured using the Code Composer Setup utility.

Terminal Emulator Command Line Switches
The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

Multiple instances of the terminal emulator may be invoked simultaneously in order to support installations utilizing multiple target boards. Instances of the terminal emulator, after the first loaded instance must be configured via command line switches in order to properly communicate with their associated target.

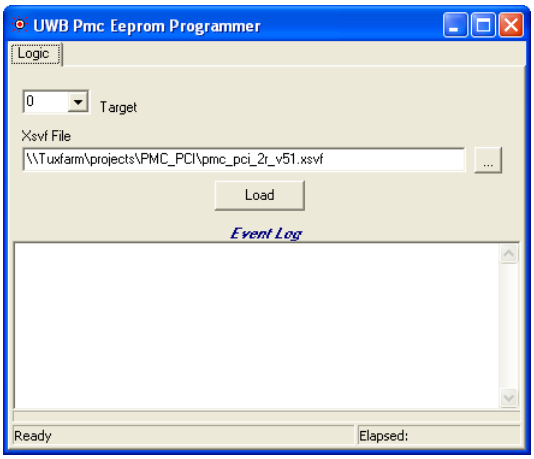
-board *boardtype* - Use the -board switch to force an instance of the terminal emulator to communicate with a specific type of target board, boardtype. Supported board types are those configured using the Code Composer Setup utility, such as "C64xx Rev 1.1 XDS560 Emulator".

-**cpu** *cputype* - Use the -cpu switch to force an instance of the terminal emulator to communicate with a specific CPU on a target board. Supported CPU types are those configured using the Code Composer Setup utility, such as “CPU_1” or “CPU_A”.

-**f** *filespec* - Use the -f switch to force the terminal emulator to load and run the specified COFF file. The “*filespec*” field should be a standard Windows file specification, including both the path and file name as a unit, to allow the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This field is particularly useful in situations where the the terminal emulator is “shelled to” from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

Baseboard-specific Applets

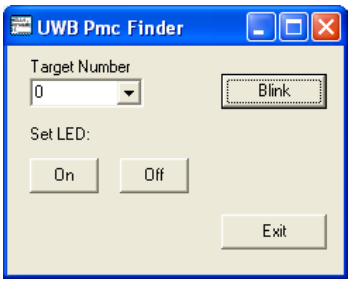
EEProm (EEProm.exe)



The baseboard features two logic devices. One controls the analog hardware and the interface to the DSP. This logic can be modified by the user, and must be loaded by the user with an image on each session.

The second device performs the baseboard enumeration and PCI interface and has a ROM so that it can function at power up. The EEProm applet is designed to allow field-upgrades of this PCI logic firmware on the baseboard. The utility permits an embedded firmware logic update file to reprogrammed into the module Flash ROM, which stores the "personality" of the board. Complete functionality is supplied in the application's help file.

Finder (Finder.exe)
The Finder is designed to help correlate board target numbers against PCI slot numbers in systems employing multiple boards.



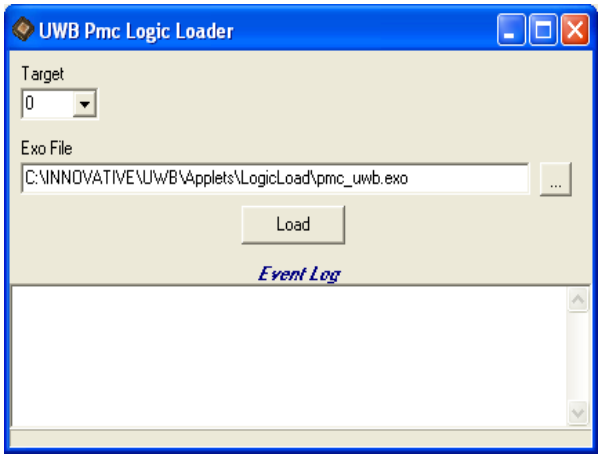
Target Number
Select the Target number of the board you wish to identify using the Target Number combo box.

Blink
Click the Blink button to blink the LED on the board for the specified target. It will continue blinking until you click Stop.

On/OFF
Use the On and Off buttons to activate or deactivate (respectively) the LED on the baseboard for the specified target. When you exit the application, the board's LED will remain in the state programmed by this applet.

Logic Loader (LogicLoader.exe)

The logic loader applet is used to deliver known-operational logic images to the user logic device on the baseboard. The user logic must be loaded once per session, as the logic part is cleared on bus reset or power up.



The utility may be used to configure the firmware either through its command line interface or from its GUI Windows user interface. The former is often convenient during PC boot-up to install a standard logic file. Place a short cut with the command line option set into the Windows Startup folder to execute the program when the system is started.

This application supports configuration of the onboard Virtex logic device from an EXO file produced by popular logic design tools (including Xilinx's). It is essential that the Virtex be programmed before using related applications, since some of the baseboard peripherals are dependent on the personality of the configured logic.

Connector Pinout and Physical Information

P25M Connector Pinouts

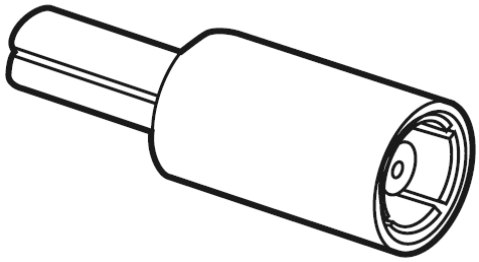
Analog Input/Output Connectors

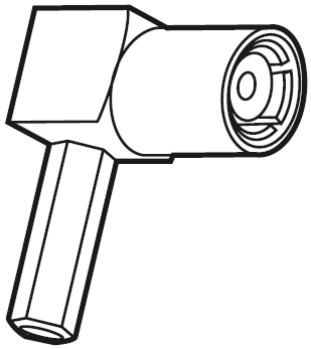
Connector Types:	50 ohm SMB , right angle
Number of Connections:	2 : inner pin is analog signal, outer shield is analog ground.
Mating Connector:	AMP P/N 413985-3 (straight) or AMP P/N 414002-7 (right angle)

Analog	Connector
Input 0	J1
Input 1	J2
Input 2	J3
Input 3	J4
Output 0	J7
Output 1	J8
Output 2	J9
Output 3	J10

J6 – External Clock Input/Output

Connector Types:	50 ohm SMB, Verticle
Number of Connections:	2 : inner is clock, outer shield is digital ground.
Mating Connector:	AMP P/N 413985-3 (straight) or AMP P/N 414002-7 (right angle)





J5 – External Sync/Trigger

Connector Types:	50 ohm SMB , Verticle
Number of Connections:	2 : inner is sync/trigger, outer shield is digital ground.
Mating Connector:	AMP P/N 413985-3 (straight) or AMP P/N 414002-7 (right angle)

JP6, JP7 – DSP McBSP Serial Port Connectors

Connector Types:	2 mm double row header
Number of Connections:	10
Mating Connector:	Samtec SQT style (for board-board applications)

The following table gives the pin numbers and functions for the JP6 (McBSP 0) and JP7 (McBSP 1) connectors. Pin functions of JP6 are identical to those of JP7 except where noted.

Table 31. DSP Serial Port Connectors

Pin Number	JP6 Function	JP7 Function	Direction (from P25M)
1	CLKS0	CLKS1	I
2	FSR0	FSR1	I/O
3	CLKR0	CLKR1	I/O
4	FSX0	FSX1	I/O
5	CLKX0	CLKX1	I/O
6	Digital 3.3V	Digital 3.3V	Power
7	DR0	DR1	I
8	-	-	-
9	DX0	DX1	O
10	Digital Ground	Digital Ground	Power

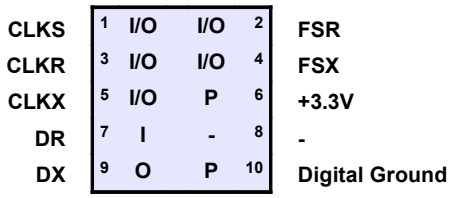


Figure 21. JP6 and JP7 DSP McBSP Serial Port Connector Pinouts

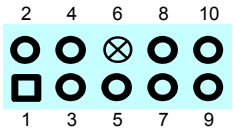


Figure 22. JP6 and JP7 DSP McBSP Serial Port Connector Pin Orientation

JP3 – Digital I/O Connector

Connector Types:	50 pin 0.1” double row shrouded male header center polarized
Number of Connections:	50
Mating Connector:	AMP P/N 1-746285-0

Digital I/O bit 0	1	I/O	I/O	2	Digital I/O bit 1
Digital I/O bit 2	3	I/O	I/O	4	Digital I/O bit 3
Digital I/O bit 4	5	I/O	I/O	6	Digital I/O bit 5
Digital I/O bit 6	7	I/O	I/O	8	Digital I/O bit 7
Digital I/O bit 8	9	I/O	I/O	10	Digital I/O bit 9
Digital I/O bit 10	11	I/O	I/O	12	Digital I/O bit 11
Digital I/O bit 12	13	I/O	I/O	14	Digital I/O bit 13
Digital I/O bit 14	15	I/O	I/O	16	Digital I/O bit 15
Digital I/O bit 16	17	I/O	I/O	18	Digital I/O bit 17
Digital I/O bit 18	19	I/O	I/O	20	Digital I/O bit 19
Digital I/O bit 20	21	I/O	I/O	22	Digital I/O bit 21
Digital I/O bit 22	23	I/O	I/O	24	Digital I/O bit 23
Digital I/O bit 24	25	I/O	I/O	26	Digital I/O bit 25
Digital I/O bit 26	27	I/O	I/O	28	Digital I/O bit 27
Digital I/O bit 28	29	I/O	I/O	30	Digital I/O bit 29
Digital I/O bit 30	31	I/O	I/O	32	Digital I/O bit 31
-	33	-	-	34	-
-	35	-	-	36	-
Ext Digital Read Enable	37	I	-	38	-
-	39	-	-	40	-
-	41	-	-	42	-
-	43	-	-	44	-
-	45	-	-	46	-
-	47	-	-	48	-
+5V (250mA max)	49	-	-	50	Ground

Figure 23. Digital I/O Connector Pinout

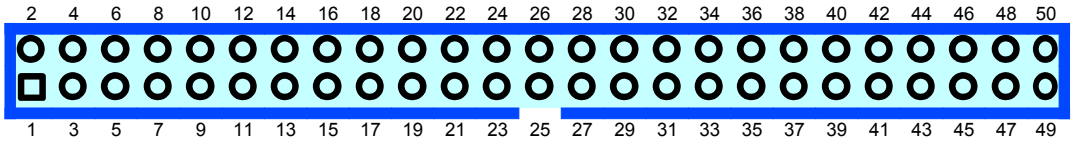


Figure 24. JP3 Digital I/O Connector Pin Orientation

JP4 – UD Digital I/O Connector

Connector Types:	40 pin 0.1” double row shrouded male header center polarized
Number of Connections:	40
Mating Connector:	AMP P/N 111810-9

Digital I/O bit 0	1	I/O	-	2	Ground
Digital I/O bit 1	3	I/O	-	4	-
Digital I/O bit 2	5	I/O	-	6	-
Digital I/O bit 3	7	I/O	-	8	-
Digital I/O bit 4	9	I/O	-	10	-
Digital I/O bit 5	11	I/O	-	12	-
Digital I/O bit 6	13	I/O	I	14	UD Read Enable
Digital I/O bit 7	15	I/O	-	16	-
Digital I/O bit 8	17	I/O	I/O	18	Digital I/O bit 31
Digital I/O bit 9	19	I/O	I/O	20	Digital I/O bit 30
Digital I/O bit 10	21	I/O	I/O	22	Digital I/O bit 29
Digital I/O bit 11	23	I/O	I/O	24	Digital I/O bit 28
Digital I/O bit 12	25	I/O	I/O	26	Digital I/O bit 27
Digital I/O bit 13	27	I/O	I/O	28	Digital I/O bit 26
Digital I/O bit 14	29	I/O	I/O	30	Digital I/O bit 25
Digital I/O bit 15	31	I/O	I/O	32	Digital I/O bit 24
Digital I/O bit 16	33	I/O	I/O	34	Digital I/O bit 23
Digital I/O bit 17	35	I/O	I/O	36	Digital I/O bit 22
Digital I/O bit 18	37	I/O	I/O	38	Digital I/O bit 21
Digital I/O bit 19	39	I/O	I/O	40	Digital I/O bit 20

Figure 25. JP4 UD Digital I/O Connector Pinout

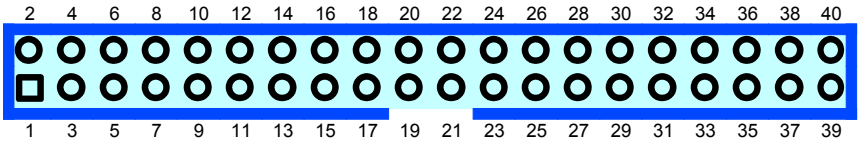


Figure 26. JP4 UD Digital I/O Connector Pin Orientation

JP8 – DSP JTAG

Connector Types:	14 pin 0.1” double row shrouded male header center polarized
Number of Connections:	14
Mating Connector:	AMP P/N 746285-2

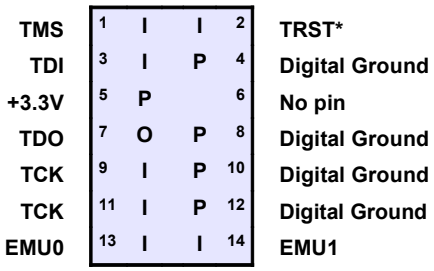

TMS 1 I I 2 TRST*
TDI 3 I P 4 Digital Ground
+3.3V 5 P 6 No pin
TDO 7 O P 8 Digital Ground
TCK 9 I P 10 Digital Ground
TCK 11 I P 12 Digital Ground
EMU0 13 I I 14 EMU1

Figure 27. JP8 DSPJTAG Connector Pinouts

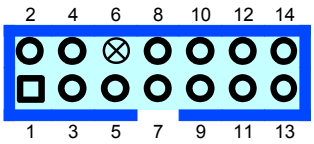


Figure 28. JP8 DSP JTAG Connector Pin Orientation

JP5 – FPGA JTAG Connector

Connector Type:	14 pin 2mm double row male header
Number of Connections:	14
Mating Connector:	AMP P/N 111623-3

Ground	1	P	P	2	3.3V
Ground	3	P	I	4	TMS
Ground	5	P	I	6	TCK
Ground	7	P	O	8	TDO
Ground	9	P	I	10	TDI
Ground	11	P		12	No Connect
Ground	13	P		14	No Connect

Figure 29. JP5 FPGA JTAG Connector Pinouts

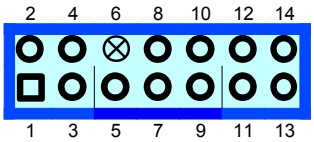
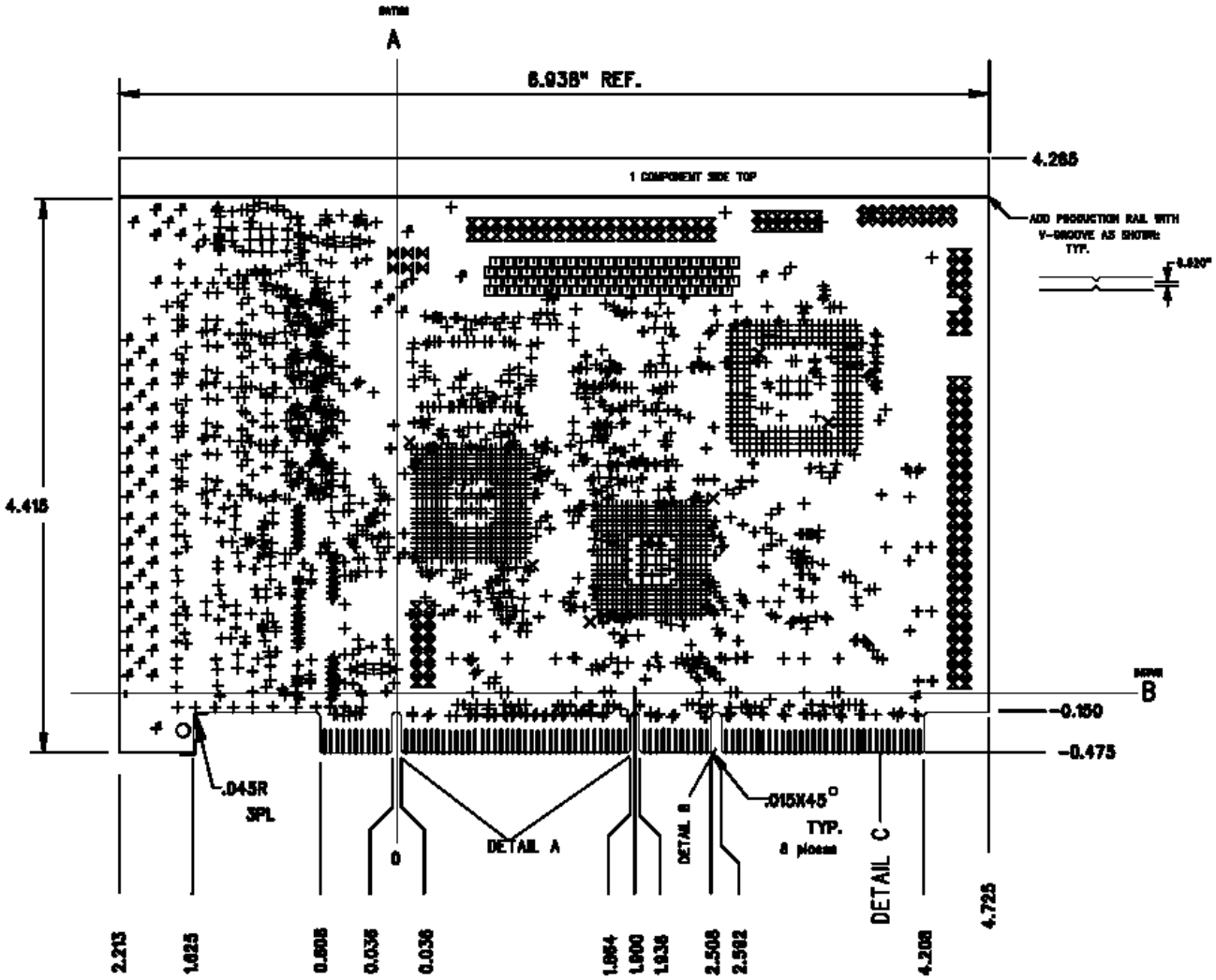
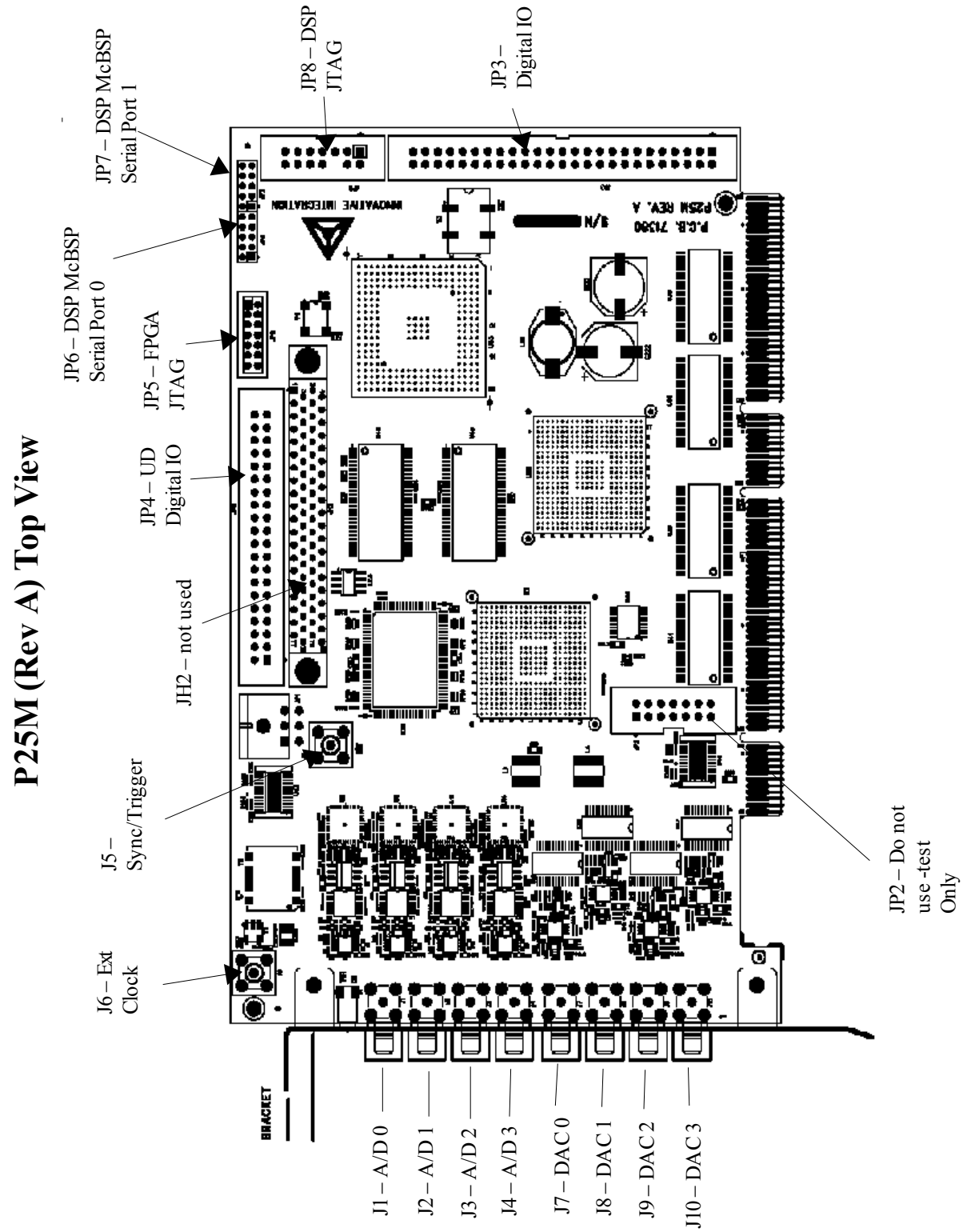


Figure 30. JP5 FPGA JTAG Connector Pin Orientation

Board Layout

A dimensional schematic and the board layout is displayed below. Please review these drawings to familiarize yourself with the circuit board's configuration and connections




P25M (Rev A) Top View
JP7 – DSP McBSP Serial Port 1
JP8 – DSP JTAG
JP3 – Digital IO
JP6 – DSP McBSP Serial Port 0
JP5 – FPGA JTAG
JP4 – UD Digital IO
JH2 – not used
J5 – Sync/Trigger
J6 – Ext Clock
JP2 – Do not use -test Only
BRACKET
J1 – A/D 0
J2 – A/D 1
J3 – A/D 2
J4 – A/D 3
J7 – DAC 0
J8 – DAC 1
J9 – DAC 2
J10 – DAC 3